Incremental Reasoning for Object Oriented Systems

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

Abstract

Inheritance and polymorphism are key mechanisms of the object-oriented approach that enable designers to develop systems in an incremental manner. In this paper, we develop techniques for *reasoning incrementally* about the behavior of such systems. A derived class designer will be able, using the proposed approach, to arrive at the richer behavior that polymorphic methods inherited from the base class will exhibit in the derived class, without reanalyzing the code bodies of these methods. The approach is illustrated by applying it to a simple case study.

Keywords and phrases: Incremental design, Incremental reasoning, Behavior of polymorphic methods.

1 Introduction and Motivation

Much of the power of the OO approach derives from the key notions of *inheritance* and *polymorphism*. Given an existing *base class* B, a designer can use inheritance to build a new *derived class* D that extends B. Some of the methods of B are redefined in D while others are inherited unchanged. Polymorphism[1] ensures that not just the methods redefined in D, but also other methods, these being the *polymorphic methods*, that invoke the redefined methods exhibit enriched behavior even though the polymorphic methods themselves are inherited unchanged from B. Inheritance and polymorphism were two of *Simula*'s [DN66] fundamental contributions that have revolutionized software design. But if we are to be able to exploit the full potential of inheritance and polymorphism, we must not only be able to *build* systems incrementally, but also to *reason* about their behavior incrementally. Our goal is to investigate the problems involved in such incremental reasoning and to develop techniques to address them.

What information about the base class B does the designer of the derived class D need in order to reason incrementally about the behavior of D? Suppose $t()$ is a method of B and that it invokes another method $h()$ of B, and suppose $h()$ is redefined in the derived class D. If $t()$ is applied to an object of type D, the $h()$ that will be invoked during this execution of $t()$ will be the one defined in D (rather than the one in B). In a sense, the polymorphic method provides the

[1] In this paper, by *polymorphism* we will mean the *subtype polymorphism* of [CW85], implemented using run-time dispatch in standard OO languages.

pattern, or template, of the calls to the methods that are intended to be redefined as needed in the derived class, while the template itself is inherited unchanged. It is for this reason that polymorphic methods are called *template methods* in the design patterns literature [GHJV95], the methods they invoke being called *hook methods*. If B includes methods[2] such as $t()$, the designer of D not only needs to reason about the behavior of the methods she defines or redefines in D, but also about the modified behavior of the polymorphic methods of B resulting from redefinitions of methods that they invoke. One possibility would be for this designer to *reanalyze* the behavior of the body of $t()$ appealing, during this reanalysis, to the modified behavior of the redefined methods. While this would work, it is clearly not an incremental approach. Thus the central question we are interested in is the following:

> What information should we include in the (base-class) specification of the template method so that a derived class designer can, in a sense, "plug-into" this specification, the behaviors of the hook methods as defined in the derived class, to arrive at the enriched behavior that the template method would exhibit (when applied to instances of the derived class), without having to reanalyze the body of the template method?

Note that reanalysis of the template method bodies is not only undesirable, it may even be impossible if, for example, the template method is part of a base class that was purchased from a software vendor who, for proprietary reasons, did not provide access to the source code.

We will see the full details of our answer to this question later, but the key is to include, in its specification, information about which hook methods $t()$ invokes, the order it invokes them in, the arguments passed to the hook methods in these calls, etc. In order to provide this type of information, we will make use of a *trace*, denoted by the symbol τ (or sometimes τ_t), to record the hook method calls that $t()$ makes. The specification of $t()$, in particular its post-condition, will give us information not only about the final values of the member variables when $t()$ finishes but also about the value of τ, i.e., information about the identity of the hook methods $t()$ invoked during its execution, the values of the arguments it passed in these calls, etc. As we will see, the derived class designer can then plug into this specification, the behavior of the redefined hook methods to arrive at the corresponding new behavior of $t()$.

There is one important requirement that these redefinitions must satisfy. Suppose that $h()$ is one of the hook methods $t()$ invokes. In arriving at the specification of $t()$ by analyzing its code in the base class, we would have made some assumptions about the effects of the call(s) to $h()$ contained in the body of $t()$. Typically, these would correspond to the behaviors exhibited by $h()$ as defined in the base class and (presumably) specified in the base class specification of $h()$. Unless the redefinition of $h()$ in the derived class satisfies its base class specification, this analysis of $t()$ may no longer be valid, and we would be forced to reanalyze the body of $t()$. Since we want to avoid such reanalysis, we will require the redefinition of $h()$ in the derived class to satisfy its base class specification.

Such a requirement is, in fact, not new to our work. It is the essential idea underlying the work on *behavioral subtyping* [Ame91, LW94, DL95]. Informally, a class A is a behavioral subtype[3]

[2] In *Simula* and *C++*, $h()$ must have been flagged as *virtual*, else the $h()$ that is invoked during the execution of $t()$ would be the one defined in B. In languages like *Java* and *Eiffel*, all methods are virtual unless explicitly declared *final*. For concreteness, we occasionally use language-specific terminology but our approach is not language-specific. Note also that we use the terms 'method' and 'function' interchangeably.

[3] Although from a formal point of view *class* and *type* are distinct notions, in most standard OO languages, as well as in much of standard OO practice, the two notions are identified. Hence in this paper we will use the two terms interchangeably. More importantly, we will only be interested in the notions of *behavioral* subtype/subclass based on

of another class B if the behavior exhibited by objects that are instances of A is in some sense consistent with behaviors allowed by the specification of class B, in other words, if the methods of class A satisfy the specifications of the corresponding methods of B. If A is a behavioral subtype of B, then any reasoning that we may have performed on a piece of code that includes calls to methods of B will continue to be valid if these calls are instead dispatched to the corresponding methods defined in A since in any such reasoning, we could only have appealed to the specifications of the methods in B and the methods defined in A satisfy these specifications. In our case, we want to be sure that whatever conclusions we have arrived at about the behavior of the template method $t()$, on the basis of the base class specifications of the hook method $h()$, continue to be valid in the derived class, so we must require that the derived class definition of $h()$ satisfy its base class specification.

What is new about our work is that, if this requirement is satisfied, then the "plugging-in" process we outlined above will allow us to arrive at the *richer* behavior that $t()$ acquires as a result of the redefinition of $h()$. Thus our work is in a sense a key extension of the behavioral subtyping approach: behavioral subtyping ensures that what we have already concluded (from the analysis of $t()$ in the base class) continues to hold following the redefinition of the hook methods in the derived class; our work allows us to reason about the richer behavior of $t()$ resulting from this redefinition. Since the very *raison d'être* of polymorphism is the ability to enrich the behavior of the polymorphic methods by suitable redefinitions of the hook methods, it is essential that the reasoning system enable us to reason about this enriched behavior. In Section 6, we will consider other related work in some detail.

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

- It identifies the key problems involved in specifying precisely the behavior of template methods and hook methods and in arriving at the derived-class behavior of a template method on the basis of its base class specification.
- It develops an incremental reasoning technique to allow the base class designer to specify the behavior of the methods of her class, and to allow the derived class designer to plug-in information about the hook methods re-defined in the derived class into the bass-class-specifications of the template methods, to arrive at the derived-class behavior of these methods.
- It illustrates the reasoning technique by applying it to a simple case study.

The rest of the paper is organized as follows: In the next section, we introduce a simple OO language fragment focused on polymorphism. In the third and fourth sections, we develop our incremental reasoning systems for specifying and verifying the behaviors of programs written in this language. The fifth section presents a simple case study to illustrate our reasoning technique. The sixth section considers related work. The seventh section reiterates the importance of an incremental reasoning system for dealing with polymorphism, summarizes our approach to such a system, and discusses possible extensions.

---

the behaviors of the methods of the classes in question, not in syntactic notions of subtype/subclass based on the signatures of the methods.

2 Language and System Model

The qualification of a method as virtual in *Simula* or *C++*, and the complementary qualification of a method as final in *Java* or *Eiffel*, allow the compiler to determine whether or not run-time dispatching must be used in dealing with that method. For reasoning about the behavior of the methods, a more useful characterization is in terms of *hook* methods and *template* methods. Given that these notions were introduced in order to talk about the designs underlying particular OO systems, it should not be surprising that they are also useful in reasoning about the behavior of such systems. In this section we introduce a simple language notation and model that characterize methods in these terms; in the next two sections we will present our reasoning technique in terms of this model. The (partial) BNF grammar for our simple language appears in Figure 1; ε in these productions denotes the empty string; note also that the symbols "{" and "}" that appear in the productions are terminal symbols (rather than extended BNF symbols indicating repetition of the enclosed constructs).

$$
\begin{array}{rcl}
\langle class \rangle & ::= & \textsf{class}\ \langle id \rangle\ \{\ \langle variables \rangle\ \langle constructor \rangle\ \langle methods \rangle\ \} \\
& | & \textsf{class}\ \langle id \rangle\ :\ \langle id \rangle\ \{\ \langle variables \rangle\ \langle constructor \rangle\ \langle methods \rangle\ \} \\
\langle variables \rangle & ::= & \varepsilon \mid \langle variable \rangle\ \langle variables \rangle \\
\langle variable \rangle & ::= & \langle simple\ type \rangle\ \langle id \rangle; \\
\langle constructor \rangle & ::= & \langle id \rangle\ (\langle parlist \rangle)\ \{\langle stmts \rangle\} \\
\langle methods \rangle & ::= & \varepsilon \mid \langle method \rangle\ \langle methods \rangle \\
\langle method \rangle & ::= & \langle method\ kind \rangle\ \langle id \rangle\ (\langle parlist \rangle)\ \{\langle stmts \rangle\} \\
\langle method\ kind \rangle & ::= & \textsf{h-method} \mid \textsf{t-method} \mid \textsf{ht-method} \mid \textsf{nht-method}
\end{array}
$$

Figure 1: (Partial) Grammar for simple OO language

The following points should be noted:

1. A base class definition specifies the name of the class, the member variables of the class, the constructor function, and the methods of the class. A derived class definition, in addition to the above, also specifies the name of the class it inherits from; note that we consider only single inheritance.

2. We assume that all member variables are *protected*, i.e., accessible to the derived class but not to client code; we also assume that all methods are *public*. Hence there are no keywords such as private or protected.

3. Each class has a (single) constructor. The name of the constructor will, as usual, be the same as the name of the class. When an instance of a derived class is constructed, the base class constructor is executed first, then the derived class constructor. Classes do not have destructors.

4. A method may be a *hook method* (h-method), a *template method* (t-method), a *hook-template method* (ht-method), or a *non-hook-template method* (nht-method). Run-time dispatching is done for h-methods and ht-methods but not for t-methods or nht-methods. h-methods and nht-methods may invoke only nht-methods; t-methods and ht-methods may invoke h-methods, ht-methods, and nht-methods.

5. Only h-methods and ht-methods may be redefined in a derived class; t- and nht-methods must be inherited unchanged. When a method is redefined, no changes may be made in the number and types of parameters it expects.

6. All member variables of a class are of simple types such as integer, boolean, etc. So an object will not contain references to other objects. The problem with allowing references to other objects is that this can lead to *aliasing* which presents some well-known problems when reasoning about behavior; since these problems are not directly related to inheritance and polymorphism which is the focus of our work, we feel it is appropriate to eliminate aliasing from the picture.

7. The parameters (other than the self object) to a method are of simple types and are all passed by-value-result. Here again allowing for passing by-reference could lead to aliasing which we wish to avoid.

Our h- and ht-methods are like the *virtual / non-final / non-frozen* methods of *Simula, C++, Java* or *Eiffel* respectively, while the t- and nht-methods are like *non-virtual / final / frozen* methods. *Simula* and *C++* allow non-virtual methods to be redefined in the derived class but such redefinitions have no effect on base class (template) methods that invoke them; we could have similarly allowed our nht-methods to be redefined without this having an effect on the t- and ht-methods that invoke them; the changes in our reasoning technique to deal with this would be straightforward. Alternately, and more importantly, we could have treated *all* methods as ht-methods. While this would be general, it would also make the reasoning task unnecessarily complex since ht-methods are the most difficult to reason about. This is similar to a base class designer flagging appropriate methods as *final*, rather than leaving, in the name of generality, every method open to redefinition in the derived classes. One point of terminology: henceforth, we will use the term '*hook method*' to mean 'h-method or ht-method', since these are the two kinds of methods in our language that can be used to serve the role that hook methods are intended to serve; similarly, we will use '*template method*' to mean 't-method or ht-method'.

Most OO languages allow the hook methods to be *abstract* in the base class; indeed, in *Simula*, a method that is defined in the class cannot be flagged as virtual. For simplicity, we do not allow such methods in our language fragment but our reasoning technique can deal with such methods, as well as with *Java*-type *interfaces* where all the methods are abstract, in a natural manner. One other point worth noting is that a compiler for the language could easily ensure that the conditions on which types of methods may be invoked by a method of a given type are indeed satisfied (or, if not, produce appropriate error messages), and ensure, in the object code, that run-time dispatching is used for the appropriate types of methods; this is no different than a *Simula* or *C++* compiler ensuring that run-time dispatching is used for *virtual* methods but not for non-virtual methods.

We conclude this section with an Account class written in our language notation. This class, in Figure 2, will serve as the base class for our case study later in the paper where we will demonstrate the application of our reasoning technique. Account has a single member variable balance that maintains the current balance in the account; the derived classes we define later will introduce additional variables. The deposit() and withdraw() operations update balance in the expected manner; these are h-methods and will be redefined in the derived class(es) to provide richer behavior. The getInfo() operation returns, as a string, the current balance in the account; note that string() returns the string representation of the value of its (integer) argument. getInfo() is also an h-method and will be redefined in the derived class; in fact, it is via this redefinition that we will be able to see, so to speak, the enriched behaviors of the other operations. It is with an eye toward this redefinition that we have defined getInfo() to return a result of type string (rather than int).

The template method that will invoke these h-methods is processTransSeq(). This method will allow us (i.e., the client code) to process a sequence of transaction requests, each request being

```
class Account {
    int balance;  // current balance

    Account(int b) { balance := b; }
    h-method deposit(int amt) { balance := balance + amt; }
    h-method withdraw(int amt) { balance := balance − amt; }
    h-method string getInfo() { return string(balance); }
    t-method processTransSeq( string transs, string results) {
        results := ⟨⟩;
        while( transs ≠ ⟨⟩ )
          {  nextReq := NextTrans( transs ); transs := RestTrans( transs );
             trans := TransName( nextReq ); amount := Amount( nextReq );
             if( trans == "deposit" ) { deposit( amount ); }
             if( trans == "withdraw" ) { withdraw( amount ); }
             if( trans == "printInfo" ) { results += "<";
                                          results += getInfo(); results += ">"; } }
    }
}
```

Figure 2: Class *Account*

for one of deposit, withdraw, or printInfo transactions. processTransSeq() has two string arguments, transs which will contain all the transaction requests, and results via which the method will return the result. processTransSeq() repeatedly reads the next transaction request from transs and processes it. In order to avoid getting involved with issues of string manipulation, we make use of a set of functions (whose definitions we omit) that allow us to extract individual transaction requests and conveniently manipulate them; thus NextTrans(transs) is the first transaction in transs; RestTrans(transs) is the string consisting of all the remaining transactions (beyond the first one); TransName(nextReq), where nextReq is a single transaction request, is the name of the transaction ("deposit", "withdraw", or "printInfo"); and Amount(nextReq), is the amount involved in the transaction (0 if the type of transaction is "getInfo")[4].

If the next transaction requested is "deposit" or "withdraw", processTransSeq() invokes the corresponding operation. If the transaction requested is "printInfo", processTransSeq() calls getInfo() and appends the returned result to results (enclosing this inside a pair of angle brackets, "<" and ">" to separate this result from the previous result in results); note that "+=" is the string append operator. This means that depending on the derived class design, when this transaction is processed, appropriate information about that particular type of account, as implemented in the (re-)definition of getInfo() in the derived class, will be appended to results. The key reasoning questions are, what information do we include in the base class specification of processTransSeq(), and how, from this specification and the derived class behaviors of the h-methods, can the derived class designer arrive at this richer behavior of processTransSeq(), as exhibited in the value it returns in results? We will see the answers to these questions in the case study section.

[4]Good OO design principles suggest that it would probably make sense to introduce an auxiliary class, Transaction, into which methods such as NextTrans() can be collected but in the interest of space, we will not do so. Note also that we have omitted the declarations of local variables, nextReq, trans, and amount, in processTransSeq().

3 Reasoning About the Base Class

Consider a base class B. The specification of B will consist of an invariant I_b and specifications for each of its methods. I_b, an assertion over the state, i.e., the member variables of B, will be satisfied at the start and end of execution of each method. Next, consider the various kinds of methods. Suppose $n()$ is an nht-method. Its specification will be of the usual form:

$$\langle pre.n(), post.n() \rangle \tag{1}$$

where the pre-condition $pre.n()$ is an assertion over the state, and the parameters passed to $n()$ at the time that $n()$ starts execution, and the post-condition $post.n()$ is an assertion over the state and the parameters to $n()$ at the time it starts execution and at the time it finishes execution. In the post-condition, we will use the *OCL* [WK99] notation *x@pre* to refer to the value of the variable x at the time $n()$ starts, and x to refer to its value when $n()$ finishes execution.

The specification of h-methods is similar. The difference between these two types of methods will show up when we consider derived classes in the next section. For nht-methods, we will essentially inherit the specification from the base class since these methods cannot be redefined in the derived class; for h-methods, we will either inherit the base class specification if the method is not redefined, or come up with an appropriate new specification if the method is redefined.

Next consider a t-method $t()$. We will associate two specifications with $t()$. The first, its *functional-* or *f-specification*, will be similar to (1) and will specify the effect of of $t()$ on member variables of B and the parameters of $t()$. The second, its *enrichment-* or *e-specification*, will be for use by the derived class designer and will include information about invocations of hook methods. We will use τ, the *trace* (or *sequence*), to record this information, and the e-specification will give us information about the value of τ:

$$\begin{array}{ll} \textit{F-specification:} & \langle pre.t(), vbpost.t() \rangle \\ \textit{E-specification:} & \langle epre.t(), epost.t() \rangle \end{array} \tag{2}$$

At its start, $t()$ has not yet invoked any h- methods so τ at that point will be the empty sequence ε. $epost.t()$ will give us information about the values of the member variables of B, of $t()$'s parameters, and of course about the value of τ when $t()$ finishes execution. Thus the relation that must hold between the assertions of the f- and e-specifications is as follows:

$$\begin{array}{ll} epre.t() & \equiv (pre.t() \wedge (\tau = \varepsilon)) \\ epost.t() & \Rightarrow post.t() \end{array} \tag{3}$$

What information concerning calls to *hook methods* (i.e., h- and ht- methods) that $t()$ makes should we include in τ? A few examples will help us answer this question. Suppose B consists of just two methods, $t()$ and an h-method $h1()$. Suppose $h1()$, as defined in B, makes no changes to the values of any of the member variables of B. Suppose in a derived class D we introduce a new (integer) variable i and redefine $h1()$ to increment i by 1 (and leave the other member variables, inherited from B, unchanged). If $t()$ is applied to an instance of D, during this execution of $t()$, calls to $h1()$ will be dispatched to the one defined in D, and hence this call to $t()$ will increment i (which is a component of the object $t()$ is applied to since this object is an instance of D) by an amount equal to the number of times $t()$ invokes $h1()$. In order to enable the derived class designer to arrive at the value that i will be incremented by during such an execution of $t()$ *without reanalyzing the body of* $t()$, the base class specification of $t()$ would therefore have to include information about how many times $t()$ invokes hook methods.

It is easy to see that this alone is not sufficient in general. Suppose in this example that there were *two* h-methods $h1()$ and $h2()$, and that $t()$ invokes each of them several times. Suppose $h1()$ is redefined in D to increment i by 1 as before, and $h2()$ is redefined to increment i by 2. Then, in order to know what effect an execution of $t()$ (applied to an instance of D) will have on i, we need to know how many times $t()$ invokes $h1()$ and how many times it invokes $h2()$, rather than just the combined total of the two. But this is also insufficient in general. Suppose that two variables i, j are introduced in D and that while $h1()$, as before, increments i by 1 each time it is invoked, $h2()$ does not change i but increments j by the current value of i, i.e., the value that i had at the time of this invocation of $h2()$. Then the effect that an execution of $t()$ has on j will depend not only on how many times $h1()$ and $h2()$ are invoked but also on how these invocations are *interleaved*; for example, if $t()$ were to invoke $h1()$ twice and $h2()$ once during its execution, then during this execution the value of j would increase by $i@pre$, or $i@pre+1$, or $i@pre+2$, where as noted earlier $i@pre$ denotes the value of i at the start of this execution of $t()$, depending respectively on whether $t()$ invokes $h2()$ before calling $h1()$, or after the first call to $h1()$, or after the second call to $h1()$.

Hence we need to be able to provide information about the *order* of the calls $t()$ makes to the hook methods. This is still insufficient. Suppose, we revise the example so that $h2()$ (as defined in D) increments j not by i but by $(i + k)$ where k is a member variable of B. Now it is quite possible that $t()$ has changed the value of k before calling $h2()$ and that it will change it further once the call from $h2()$ returns. In this case, to arrive at the effect that $t()$ will have on j without reanalyzing its body, we will need to know what value $t()$ has left in k immediately before the call to $h2()$. In general, we would need to know the entire 'state', i.e., the values of all member variables of B, before each call $t()$ makes to a hook method, and this information will have to be recorded in τ. It turns out that we also need to record the state immediately *following* the return from each call to a hook method; this is because it is possible that the hook method might, according to its specification, assign one of two different values to one of the member variables (of B) and what $t()$ does following the return from the hook method, including what other hook methods it calls, might depend on this value; so in order to be able to relate the values in these variables to what these later calls might do (including, in particular, assigning values to other member variables, some of which might be introduced in the derived class), we need to record in τ the state following each hook method call. Finally, if the hook method receives any additional parameters, we also need to record the values of these arguments and the results returned by the hook method since, as in the case of the values of member variables of B, what $t()$ does following the return will, in general, depend on these results.

To record all this information, we will use the following structure for the sequence τ. Each element of τ will represent one call to a hook method and the corresponding return. As noted earlier, at the start of the execution of $t()$, τ will be the empty sequence ε. Suppose at some point in this execution the current state, i.e., the values of all member variables of B, is σ', and $t()$ invokes an h-method $h()$, the values of the additional arguments passed to $h()$ being $\overline{aa}'$; and suppose that the state when $h()$ returns is σ, and the result values of the additional parameters are $\overline{aa}$. Then this call-return will be recorded in τ as the element:

$$(h,\ \sigma',\ \overline{aa}',\ \sigma,\ \overline{aa}) \tag{4}$$

If $h()$ were an ht- rather than an h-method, we would again record the same information in τ about a call from $t()$ to $h()$. Note that in this case, during its execution, $h()$ may in turn invoke another h-method (or ht-method), call it $g()$. Although this call to $g()$ did arise as a result of the original call that $t()$ made to $h()$, the call to $g()$ will *not* be recorded in the trace of $t()$ (it will, of course, be recorded in the trace of $h()$). If $g()$ were to be redefined in the derived class, the derived class

designer would be able, as we will see in detail in the next section, to arrive incrementally at the resulting enriched behavior of $h()$ on the basis of the e-specification of $h()$ (and the information that specification provides about the calls to $g()$ that $h()$ makes), and *then* arrive at the enriched behavior of $t()$ on the basis of the e-specification of $t()$ and the information it provides about the calls that $t()$ makes to $h()$. Thus the enrichment in the behavior of $t()$ arises because of the enrichment in the behavior of $h()$; whether that latter enrichment is due to a redefinition of $h()$ in the derived class or due to a redefinition of an h-method that $h()$ invokes is not relevant when reasoning about the enriched behavior of $t()$. In other words, the functioning of $t()$ depends only on what the call to $h()$ does and what enrichment is done to this behavior of $h()$, not *how* that enrichment is achieved, so we only need record the call to $h()$ in the trace of $t()$, not the calls to h-methods that $h()$ in turn may make.

So much for the structure of τ. In what form should information about τ be included in $epost.t()$, the e-post-condition of $t()$? One extreme approach would be to explicitly list, in $epost.t()$, all the possible values τ could have when $t()$ finishes, i.e., list all the different sequences of hook method calls that $t()$ could have gone through during its execution, and for each, provide complete information about each component of each element of τ. While this would work, doing it naively would generally be far too tedious. A better approach is to define suitable functions, the details of which may depend on the particular application, on τ, and write the specification in terms of these functions; we will see in our case study.

Further, it is usually not necessary to provide complete information about τ. This depends in part upon the kind of enrichments the base class designer expects will be made in the derived classes. If, for example, in the case of the Account class defined in the last section, we do not expect the hook methods to be redefined in such a way as to depend on the value of balance at the time that the hook method is invoked, then there is no need to include information about this in specifying the t-method processTransSeq(). On the flip-side, if the derived class designer *does* redefine a hook method in such a way that its enriched behavior critically depends on the value of balance, she would be unable to reason incrementally about the corresponding enriched behavior of processTransSeq(). We will return to this point later.

How do we show that $t()$ meets its specifications? The main problem has to do with showing that the body of $t()$ meets its e-specification because once we do that, we simply need to check that the relation specified in (3) holds in order to conclude that $t()$ meets its f-specification as well. When reasoning about the body of $t()$, we use standard axioms and rules for dealing with standard statements such as assignment and *if-else*. The one statement for which we need a new rule is call to h-method (or ht-method), to account for recording on τ, information about the call.

R1. h/ht- **Method Call**

$$p \Rightarrow (I_b \wedge pre.h(\overline{x})[\overline{x} \leftarrow \overline{aa}])$$
$$[(\exists \sigma', \overline{aa}').[\ p[\tau \leftarrow abl(\tau),\ \sigma \leftarrow \sigma',\ \overline{aa} \leftarrow \overline{aa}']$$
$$\wedge\ post.h(\overline{x})[\sigma@pre \leftarrow \sigma',\ \overline{x}@pre \leftarrow \overline{aa}',\ \overline{x} \leftarrow \overline{aa}] \wedge I_b$$
$$\wedge\ last(\tau) = (h, \sigma', \overline{aa}', \sigma, \overline{aa})\ ]\ ] \quad \Rightarrow \quad q$$

$$\{\ p\ \}\ h(\overline{aa});\ \{\ q\ \}$$

$h()$ is the method being called, $\overline{aa}$ being the (additional) arguments for this call. The first antecedent of **R1** requires us to show that if the assertion p which is the pre-condition of the call is satisfied, then I_b, the invariant of B is satisfied; and the pre-condition $pre.h()$ of the (f-)specification

of the method[5] is satisfied with the actual arguments ($\overline{aa}$) substituting for the formal parameters ($\overline{x}$); "$\leftarrow$" denotes (simultaneous) substitution of all occurrences, in the given assertion, of the variable(s) on the left side of the "$\leftarrow$" by the expression(s) on the right. The second antecedent requires us to show that we have added a new element to τ corresponding to this call and that the state at this point (and the returned values of the arguments) satisfy the post-condition of the call to $h()$. $last(\tau)$, as the name suggests, is the last, i.e. the rightmost, element of τ; $abl(\tau)$ stands for "*all but the last element of* τ" and is the sequence obtained from τ by omitting its last element. In more detail, in this antecedent, σ' denotes the state that existed immediately before the call to $h()$ and $\overline{aa}'$ the values of the arguments at that point; so this antecedent requires us to show that: if the state (and argument values) that existed immediately before the call and the trace, less its last element, satisfy the assertion that is the pre-condition of the call; and if the post-condition of the f-specification of $h()$ is satisfied with appropriate substitutions for the before- and after-states and argument values; and if the class invariant is satisfied; and if the (newly added) last element of τ consists of the name of the called method (h), the state (σ') immediately before the call, the initial value ($\overline{aa}'$) of the arguments, the state (σ) immediately after the return from h, and the final values of the arguments ($\overline{aa}$); Then it must be the case that the specified post-condition q of this call to $h()$ is also satisfied. If these two antecedents can be shown then, by appealing to the rule, we may derive the specified conclusion.

Although the rule looks rather involved, the complexity is mostly notational. It just captures the fact that the effect of the call to $h()$ is to modify the values of the member variables of B and the arguments passed to $h()$ as specified in the (functional) post-condition of $h()$, and to append an appropriate element to τ to represent the call/return. In practice, in reasoning about the body of $t()$, we encounter such a call we would typically simply write appropriate pre- and post-conditions for the call statement and check semi-formally that what these assertions say about the changes in the values of the member variables of B, the values of the arguments to $h()$, and the value of τ, are consistent with what the f-specification of $h()$ says will be the effect of the method on the members of B and the parameters to $h()$, and with recording this call/return on τ.

The final type of method is the ht-method. Suppose $ht()$ is such a method. Its specification will be similar to that of a t-method. In other words, $ht()$ will have f- and e-specifications. The former specifies the effect of an execution of $ht()$ on the member variables of B and the parameters of $ht()$, and the latter provides information also about the calls that $ht()$ makes to hook methods during its execution. The key difference with t-methods will show up when we consider derived classes. For t-methods, we will use the e-specification from the base class and arrive at its enriched behavior (and the corresponding f-specification) by appealing to the richer behavior of the hook methods it invokes. We will do the same also for ht-methods that are inherited unchanged from the base class. But if an ht-method is redefined in the derived class, we will come up with appropriate new f- and e-specifications.

Let us now briefly turn to invariants. In our system, when reasoning about the base class B, we use a standard approach to dealing with invariants. In other words, for each method $f()$, the result we establish for S, the body of $f()$ is:

$$\{\ I_b \wedge pre.f()\ \}\ S\ \{\ I_b \wedge post.f()\ \} \tag{5}$$

where I_b is the invariant for B. Further, when establishing this result, for dealing with calls in S

[5] If $h()$ is an ht-method, it will have, as we will see shortly, both an f- and an e-specification in the same manner as t-methods. But as far as $t()$ is concerned, only the functional effect of $h()$ is relevant; thus the pre- and post-assertions referred to in the antecedents of **R1** are from $h()$'s f-specification.

to other methods (either nht-methods or hook methods) of B, we must check that not only is the pre-condition of the method being called satisfied but also the invariant; and, conversely, we may assume, when the method call returns, that not only will the method's post-condition be satisfied but also the invariant. Any functions redefined in a derived class D of B will also have to maintain this invariant (since otherwise, if $f()$ were a t-method and one of the calls in S is dispatched to such a redefined method, the assumption made in establishing (5) that I_b will hold when this call returns will no longer be valid); we will formalize this requirement in the next section. One type of method we have not considered so far is constructors. Clearly, we must check that each constructor $c()$ of B is such that when it finishes execution, I_b is satisfied. The final step in reasoning about B is to ensure that it meets its *abstract specification*, intended for use by clients of B. This can be done in a standard fashion, see for example [Jon90]; inheritance and polymorphism do not add any complexity to these issues, so we will not discuss them further.

We conclude this section with a comment about our trace τ. τ is like an *auxiliary variable* of Owicki and Gries [OG76], but there are some differences. In systems such as those of [OG76], we are allowed, when reasoning about the behavior of a piece of code, to introduce as many auxiliary variables of whatever types as we wish; we also have to introduce suitable assignment statements (into the code whose behavior we are reasoning about) to update the values of the auxiliary variables at appropriate points as we wish. By contrast, in our system, τ is the only additional variable; its structure is fixed, as specified in (4); the updates to τ take place automatically with each call that $t()$ makes to a hook method; this is represented in our system by the rule **R1**. Note also that τ is *not* a member variable of the class; it only records the calls that this method $t()$ makes to h- and ht- methods during one particular execution; thus, τ is like a local variable of $t()$, initialized, as specified in (3), to ε at the start of this execution. Its purpose is not so much to help reason about the behavior of the base class B as to provide more information in the e-specification of $t()$ than can be provided using just the member variables of B. And the purpose of providing this extra information is to enable us to arrive at the richer behavior of $t()$ that results from redefinitions, in a derived class of B, of one or more of the methods that $t()$ invokes, without having to reanalyze the body of $t()$. Thus while Owicki-Gries type auxiliary variables are introduced to help in reasoning about the behavior of the piece of code under consideration, we have introduced τ to help the derived class designer to reason incrementally about the behavior of her derived class.

4 Incremental Reasoning About the Derived Class

Let D be a derived class of B. In our skeletal language, as in most standard OO languages, the designer of D may introduce new member variables in D, define entirely new methods, or redefine hook methods inherited from the base class; nht-methods and t-methods must be inherited unchanged. For methods that are newly defined in D, we use the same approach as in B. From the point of view of *incremental* reasoning, the key question is how to arrive at the richer behavior of inherited template methods without reanalyzing the body of the template method. This question will be the main focus of this section but we start our discussion with the relation between the invariants for B and D and then consider ways to reason about each type of method.

Let I_b, I_d be the invariants for B, D. Since some of the methods will be inherited unchanged from B, and since these methods require I_b to be satisfied before they start execution, we will require the following:

$$I_d \Rightarrow I_b \tag{6}$$

And in order to ensure that each method in D, including the inherited ones, leave I_d satisfied when they finish execution, we will have to impose further conditions on the specifications for the individual methods as we will see below. (6) will be part of the *behavioral subclassing* relation to be defined shortly.

Suppose $n()$ is an nht-method inherited from B. The (concrete) specification of $n()$, as a method of D, will be in terms of the overall state, i.e., the values of the member variables defined in D as well as those inherited from B. For convenience, in our discussion below, we will use σ to denote the overall state, $\sigma \downarrow b$ to denote the portion of the state inherited from B, and $\sigma \downarrow d$ the portion defined in D. Let $\langle pre.B.n(), post.B.n() \rangle$ be the specification of $n()$ in the base class B. Since the method is inherited unchanged by D, execution of $n()$ cannot change the value of any variable introduced in D, i.e., the value of $\sigma \downarrow d$ when $n()$ finishes execution will be the same as when it started. Hence, $\langle pre.D.n(), post.D.n() \rangle$, the specification of $n()$ in D, follows from its base class specification if the following conditions are satisfied:

$$(pre.D.n() \wedge I_d) \Rightarrow pre.B.n()$$

$$(post.B.n() \wedge I_b \wedge (pre.D.n() \wedge I_d)[\sigma \leftarrow \sigma@pre] \wedge (\sigma \downarrow d = \sigma \downarrow d@pre)) \Rightarrow (post.D.n() \wedge I_d) \qquad (7)$$

If $pre.D.n()$ is satisfied when $n()$ is invoked, the relation between the pre-conditions ensures that $pre.B.n()$ will be satisfied at that point. Hence, given that we have checked (when reasoning about the base class) that the body of $n()$ satisfies its base class specification, the assertion $post.B.n()$ (and I_b) will be satisfied when $n()$ finishes execution. In addition, the clause $(\sigma \downarrow d = \sigma \downarrow d@pre)$ which is essentially an abbreviation for a set of clauses that assert, for each member variable introduced in D, that its value is unchanged from its value at the start of $n()$, will also be satisfied since these variables are unaffected by $n()$. The clause $(pre.D.n() \wedge I_d)[\sigma \leftarrow \sigma@pre]$ asserts that the state, including the values of the variables introduced in D, at the time $n()$ started execution satisfies the (new) pre-condition and invariant. Note that $pre.D.n()$ may include conditions on the values of the variables introduced in D. In that case, the assertion $(\sigma \downarrow d = \sigma \downarrow d@pre)$ will allow us to carry these conditions forward to $post.D.n()$. This may be of help in showing, as required by (7), that I_d will hold at that point.

Next consider $h()$, an h-method. If $h()$ is inherited unchanged, we treat it in the same way as an nht-method. If $h()$ *is* redefined in D, the derived class designer will have to come up with a new specification, $\langle pre.D.h(), post.D.h() \rangle$, and check (or formally verify) that the redefined method satisfies this specification (as well as I_d, as required by (5)). In either case, we also need to impose a requirement of behavioral consistency with the base class specification $\langle pre.B.h(), post.B.h() \rangle$ of $h()$ since otherwise any reasoning that we have done (concerning the behavior template methods that invoke $h()$) on the basis of that specification may no longer be valid.

Definition: The derived class D is a *behavioral subclass* of its base class B if the following conditions are satisfied:

$$I_d \Rightarrow I_b$$

If $h()$ is an h-method or an ht-method, then

$$(pre.B.h() \wedge I_b) \Rightarrow (pre.D.h() \wedge I_d)$$

$$(post.D.h() \wedge I_d) \Rightarrow post.B.h() \qquad (8)$$

We require, as part of our reasoning system, that D be a *behavioral subclass*[6] of B.

[6] (8) is very similar to behavioral subtyping [LW93, LW94]; nevertheless we use a different term since behavioral subtyping is a relation that involves the abstract specifications of two classes while ours is a relation between the concrete specifications of a base class and its derived class.

Consider a call to $h()$ in a t-method, $t()$. In D, this call will be dispatched to the $h()$ defined in D; so when the call returns, $(post.D.h() \wedge I_d)$ will be satisfied and hence, by the relation between post-conditions and invariants required by (8), so will $(post.B.h() \wedge I_b)$ which is what we must have assumed when reasoning about $t()$ in the base class. Thus behavioral subclassing ensures that the reasoning we have performed in the base class about a method that calls $h()$ continues to be valid although $h()$ has been redefined in the derived class. Note also that in order for $post.D.h()$ to be satisfied when $D.h()$ finishes execution, $(pre.D.h() \wedge I_d)$ must have been satisfied at the time of the call to $h()$; the relation required by (8) between the pre-conditions and invariants, given that when reasoning in the base class about the calls in $t()$ to $h()$ we must have checked that $(pre.B.h() \wedge I_b)$ is satisfied immediately prior to each of these calls, ensures this. It is worth noting that (8) imposes severe constraints on the derived class. In particular, the relation that (8) requires between the base class pre-condition & invariant and the derived class pre-condition & invariant means that the derived-class pre-condition of $h()$ cannot impose any requirements on the values of member variables that may be introduced in D. Nevertheless, by using the *@pre* notation to refer to the values of variables at the start of $h()$, we will be able, in $post.D.h()$, to specify how $D.h()$ changes the values of variables introduced in D; and the rule **R2** will allow us to appeal to this information to arrive at the effect that $t()$ has on these variables. In more detail, we will look at each element in the trace τ of $t()$, and add, to the base-class specification of $t()$, the assertion that the states and argument values recorded in this particular element of τ satisfies the derived class post-condition of the h-method invoked. Suppose the k^{th} element of τ is (h, $\sigma 1$, $\overline{aa}1$, $\sigma 2$, $\overline{aa}2$), then:

a. We can assert $post.D.h()$ with $\sigma 1$ and $\sigma 2$ playing the roles respectively of the state immediately before the call and the state immediately after the call, and $\overline{aa}1$ and $\overline{aa}2$ being the argument values before and after the call.

b. We can assert that the D-portion of the state can change only due to calls to h/ht-methods since $t()$ was defined as part of the base class so its code cannot refer to this portion of the state. Thus if σp is the 'final state' of the *previous* element of τ, i.e., is the fourth component of the $(k-1)^{th}$ element of τ, then we must have $(\sigma p \downarrow d = \sigma 1 \downarrow d)$; similarly if σn is the 'initial state' of the *next* element of τ, i.e., is the second component of the $(k+1)^{st}$ element of τ, then $(\sigma 2 \downarrow d = \sigma n \downarrow d)$.

Rule **R2** below formalizes these ideas. The following functions and predicates on traces, trace elements and their components, etc., will be useful in expressing this formalization:

$|\tau|$: Length of τ, i.e., number of elements in τ.

$\tau[k]$: The k^{th} element of τ.

$\tau[k].hm$: The identity of the hook method called in the k^{th} element of τ.

$\tau[k].is$: The initial state, i.e., the state just before this call.

$\tau[k].fs$: The final state, i.e., the state just after the method returns.

$\tau[k].ia$: The values of the arguments passed in this call.

$\tau[k].fa$: The values of the arguments when the method returns.

$\tau \downarrow b$: Same as τ except that in each 'state' component of each element of τ, we only retain the base-class portion of the state; similarly $\tau \downarrow d$ is obtained from τ by retaining, in each 'state' component of each element of τ, only the portion of the state introduced in the derived class. Naturally, $\downarrow b$ and $\downarrow d$ operations are applicable only to traces at the derived class level.

ncbc$(\sigma i, \tau, \sigma f)$: *ncbc*() denotes "no change between calls"; i.e., the D-portion of the state does not change between calls to hook methods. σi is the initial state, i.e., the state at the start of $t()$, and σf the final state, i.e., the state at the end of $t()$. More formally:

$$\begin{array}{l} ncbc(\sigma i, \tau, \sigma f) \equiv \\ \quad ((|\tau| = 0) \Rightarrow (\sigma i \downarrow d = \sigma f \downarrow d)) \\ \quad \wedge ((|\tau| > 0) \Rightarrow ((\sigma i \downarrow d = \tau[1].is \downarrow d) \wedge (\sigma f \downarrow d = \tau[|\tau|].fs \downarrow d) \wedge \\ \qquad\qquad\qquad (\forall j : (1 \leq j < |\tau|) :: (\tau[j].fs \downarrow d = \tau[j+1].is \downarrow d)))) \end{array}$$

Since σi is the state that $t()$ starts in, the $\downarrow d$ portion of the state just before the first call recorded in τ will be same as the $\sigma i \downarrow d$ since the portion of $t()$ that precedes this call cannot have modified it. Similarly, the $\downarrow d$ portion of the state when $t()$ finishes execution will be same as the $\downarrow d$ portion of the state immediately after the last call recorded in τ. This explains the first two clauses in the case that $(|\tau| > 0)$. The third clause states that the $\downarrow d$ portion of the 'initial state' recorded in the $(j+1)^{st}$ call is the same as the $\downarrow d$ of the 'final state' recorded in the j^{th} call. If $(|\tau| = 0)$, the $\downarrow d$ portion of the final state when $t()$ finishes is the same as $\downarrow d$ of the state at the start of $t()$ since no hook methods are invoked.

ccds(τ, k): *ccds*() denotes "*c*hange (in state) recorded in the k^{th} element of τ is *c*onsistent with *d*erived-class *s*pecification (of the hook method)"; i.e., the states and argument values recorded in the k^{th} element of τ is consistent with the f-specification, in the derived class, of the hook method invoked in this element. More formally:

$$\begin{array}{l} ccds(\tau, k) \equiv (post.D.\tau[k].hm()[\overline{xx}@pre \leftarrow \tau[k].ia,\ \sigma@pre \leftarrow \tau[k].is, \\ \qquad\qquad\qquad\qquad\qquad\qquad \overline{xx} \leftarrow \tau[k].fa,\ \sigma \leftarrow \tau[k].fs] \\ \qquad\qquad \wedge I_d[\sigma \leftarrow \tau[k].fs]) \end{array}$$

ccds() asserts that the initial and final values of the arguments and initial and final states recorded in $\tau[k]$ satisfy the conditions that the derived class f-post-condition imposes on the initial and final values of its parameters and the initial and final states when the method begins and ends. *This* is what will allow us to arrive at the richer behavior of $t()$ by appealing to the richer behavior of the redefined $\tau[k].hm$ as expressed in its derived class f-post-condition.

With these preliminaries out of the way, we can present the main rule **R2** that makes it possible to reason incrementally in our system. The rule requires us to establish the specified antecedents

R2. Enrichment Rule

$$\begin{array}{l} (epre.D.t() \wedge I_d) \Rightarrow epre.B.t() \\ [epost.B.t()[\tau \leftarrow \tau \downarrow b,\ \sigma \leftarrow \sigma \downarrow b,\ \sigma@pre \leftarrow \sigma@pre \downarrow b] \\ \quad \wedge (pre.D.t() \wedge I_d)[\sigma \leftarrow \sigma@pre] \\ \quad \wedge ncbc(\sigma@pre, \tau, \sigma) \wedge (\forall k : (1 \leq k \leq |\tau|) :: ccds(\tau, k))] \quad \Rightarrow \quad (I_d \wedge epost.D.t()) \end{array}$$

$$\langle epre.D.t(), epost.D.t() \rangle$$

in order to conclude the derived-class e-specification for $t()$. The first antecedent requires us to show that if a state satisfies the derived-class pre-condition of $t()$, it also satisfies the base-class pre-condition. This is needed because when reasoning, in the base class, about what $t()$ does when it starts, we had assumed that the state satisfies the base-class pre-condition; so unless this is true, that reasoning may no longer be valid. In the base class reasoning, we also assumed that the initial state satisfies I_b; this will still be the case because in the derived class we may assume that the state will satisfy I_d at the start of $t()$, and hence, given the requirement of behavioral subclassing, the state will also satisfy I_b.

In the second antecedent, σ*@pre* and σ denote the complete (i.e., both base- and derived-class portions) initial and final states when $t()$ begins and ends execution. Since *epost.B.t*() refers only to the base class portion of the state, we replace σ and σ*@pre* in *epost.B.t*() by the $\downarrow b$ portion of these states. Similarly, we replace τ in *epost.B.t*() by $\tau \downarrow b$. In practice, these substitutions tend to require no real effort. Thus, for example, suppose x is a member variable of B and that *epost.B.t*() contains a clause $(x = x@pre + 10)$; since x is a component of both σ and $\sigma \downarrow b$ (and x*@pre* a component of σ*@pre* and σ*@pre*$\downarrow b$), nothing needs to be done, as far as this clause is concerned, to effect the substitutions. We will see this in practice in the case study later in the paper.

Thus this antecedent requires us, given the base class e-post-condition of $t()$, given that the derived class portion of the state doesn't change between calls to hook methods, and given that following each call recorded on τ, the state and argument values satisfy the derived class post-condition of the method called, to show that the derived-class invariant and the derived class e-post-condition of $t()$ are satisfied. As explained earlier, it is the assumption that the state and argument values following calls to the h/ht-methods satisfy the richer derived class specification of these methods, that allows us to arrive at a correspondingly richer post-condition *epost.D.t*() for $t()$ without having to re-analyze its body.

It maybe useful to summarize our approach for reasoning about the derived class D: We first come up with the invariant I_d for the class, the e-specification and f-specification for each t-method and each ht-method of the class, and the specification for each h-method and each nht-method. Next we check that D is a *behavioral subclass* of B, i.e., the requirements specified in (8) are satisfied. Next, we have to verify that each method satisfies its specification(s). For each method that is newly defined or is redefined in D, we use the same approach as in the base class; for the redefined methods, we also check (or have checked) that the relation, imposed by the behavioral subclassing requirement, between the method's derived class specification and its base class specification. For non-template methods inherited from the base class, as we saw in (7), the specification is the same as in the base class with the addition, in the post-condition of the method, that the method does not change the values of member variables newly introduced in D. For template methods, and this is the focus of our paper, we use the rule **R2** to (arrive at and) justify the richer e-specification for the method, and in turn use this richer e-specification to justify a correspondingly richer f-specification (as required by (3)).

One point is worth stressing: if $h()$ is an ht-method that is redefined in D, the behavioral subclassing requirement has to do with its *f-specification*, not its *e-specification*. This is because, so long as the (functional) behavior of the redefined $h()$ is consistent with its base class f-specification, the reasoning that we have done in the base class about the behavior of any (template) methods that invoke $h()$ will remain valid. The point is that the e-specification for $h()$ in the base class would have allowed us to arrive (using rule **R2**) at its richer behavior if we had inherited $h()$ unchanged but had redefined some of the hook methods it invokes; if instead we redefine $h()$ in D, then its base class e-specification is of no particular relevance in the derived class.

One important question that any axiomatic system has to address is that of soundness and (relative) completeness with respect to the operational model of the programming language/system. Because of space limitations, we will consider this question only briefly. The main question concerns the behavior of template methods since our approach to the other methods is standard. And here, one problem in establishing soundness and completeness of our system is that the trace τ that plays a central role in our axiomatic system is not part of standard operational models of OO languages; as a result, we cannot talk about the validity of our e-specifications with respect to our model.

This may seem an advantage since we would then have to worry only about the f-specifications. But the problem is that in our approach, we first establish the e-specification (with rules **R1**, **R2** being the key ones for dealing with the trace information), and then establish the f-specification by showing that the conditions specified in (3) are satisfied. Hence we cannot establish the validity of an f-specification without first showing the validity of the e-specification that the f-specification is based on. The solution is to introduce traces also into the operational model[7]. As in the axiomatic system, the trace in the model would record calls to and returns from hook methods; each such call-return would record the name of the method called, the argument values and state at the time of the call, and the argument values and state at the time of the return. With this change, it is straightforward to show that results established using our reasoning system in particular using rule **R1**, about a base class are valid in the model.

Results about a derived class, in particular those established using **R2**, are more difficult. One possible approach would be as follows: Consider the proof outline (in the base class) that established the original e-specification of the template method in question. Treat the method as a member of the derived class and develop a new proof outline; this new outline is obtained by adding, to each assertion in the original proof outline, the clauses $ncbc()$ and $ccds()$ (for all $k < |\tau|$). These clauses must hold at all points in this method (considered as a member of the derived class) for the same reason as before, that is, the member variables introduced in the derived class can change only due to the calls to the hook methods. Thus this new proof outline justifies the enriched e-post-condition that appears in **R2**, and hence shows that any result derived by using that rule[8] must be valid in the model.

So much for soundness. Now consider (relative) completeness. To show completeness, we have to show that the strongest post-condition for any method, i.e., the assertion that is satisfied only by states that can operationally arise when method finishes execution, can be established. Here again the argument is best presented in terms of proof outlines. Consider the base class. For each statement in the method, we simply use the strongest post-condition corresponding to the statement and its pre-condition. Then we can inductively argue that the resulting post-condition is indeed the strongest possible one for the entire method. Consider now a template method and its proof outline in the base class. From this, derive a new proof outline for the method in the derived class by adding the $ncbc()$ and $ccds()$ (for all $k < |\tau|$) clauses as before to each assertion in the base-class proof outline. Again we can argue inductively that the assertion specified in this proof outline at each point in the method is indeed the strongest possible assertion at that point; and hence that, using **R2**, we will be able to establish the strongest possible post-condition that applies, in the derived class, to this method.

5 Case Study

The base class for our case study is the Account class defined in Section 2, and we will consider two derived classes of Account. We start with, in Figure 3, the specifications for the constructor and the hook methods of Account. The specification of the constructor states that the balance in the constructed account is initialized to the given value. The specifications for deposit() and withdraw()

[7]This is not to suggest that anything is to be gained by introducing traces into actual implementations of OO languages. Our only purpose in introducing traces into the operational model is to bring the model closer to the reasoning system so that soundness and completeness arguments can be more easily developed. When defining a new model in this manner, we must of course ensure that as far as possible values that variables that already exist in the

$$
\begin{aligned}
&\mathit{pre}.\mathsf{Account(b)} \equiv (\mathsf{b} > 0)\\
&\mathit{post}.\mathsf{Account(b)} \equiv (\mathsf{balance} = b)\\
&\mathit{pre}.\mathsf{Account.deposit(x)} \equiv (\mathsf{x} > 0)\\
&\mathit{post}.\mathsf{Account.deposit(x)} \equiv ((\mathsf{x} = \mathsf{x}@\mathit{pre}) \wedge (\mathsf{balance} = \mathsf{balance@pre} + \mathsf{x}))\\
&\mathit{pre}.\mathsf{Account.withdraw(x)} \equiv (\mathsf{x} > 0)\\
&\mathit{post}.\mathsf{Account.withdraw(x)} \equiv ((\mathsf{x} = \mathsf{x}@\mathit{pre}) \wedge (\mathsf{balance} = \mathsf{balance@pre} - \mathsf{x}))\\
&\mathit{pre}.\mathsf{Account.getInfo()} \equiv (\mathit{true})\\
&\mathit{post}.\mathsf{Account.getInfo()} \equiv ((\mathsf{balance} = \mathsf{balance@pre}) \wedge (\mathit{string}(\mathsf{balance}) \preceq \mathit{result}))
\end{aligned}
\tag{9}
$$

Figure 3: Specification of Account class

tell us that these methods do not change the value of the parameter x, and that they update balance appropriately.

The specification of getInfo() is more interesting. Note first that in this post-condition we use *result* to refer to the value returned by this function [Mey97]. Also we assume that the $\mathit{string}(x)$ represents the string version of x; and "$\preceq$" is the *prefix* relation over strings. Thus this specification tells us that getInfo() leaves balance unchanged, and that the string representation of the balance is a prefix of the *result* returned. The result returned by Account.getInfo() is in fact *equal* to this string, but the specification allows the derived class designer to redefine getInfo() to return additional information beyond the balance in the account (while still satisfying the behavioral subclassing requirement). If our specification instead stated that the result returned by getInfo() was equal to balance, the derived class designer would be prevented, by behavioral subclassing, from implementing such enrichments. By the same token, the specifications of deposit() and withdraw() forbid the redefinition of these methods to, say, impose a transaction fee by deducting an additional amount from the balance. It is straightforward to show that the bodies of the hook methods of the Account class, as defined in Fig. 2, do satisfy the specifications in (9).

Next consider the template method processTransSeq(). It may be useful to briefly summarize the operational behavior of the code, which appears in Figure 2, of this method: The method receives a sequence of transaction requests in its first parameter transs; it extracts each transaction from transs, and invokes the corresponding method; if the transaction is printInfo, it appends to its second parameter results (whose initial value is the empty string) the result returned by the call to getInfo; and terminates after processing all the transactions in transs.

In the e-specification of this method in Fig. 4, we use a number of auxiliary functions and predicates; we start by defining these and then will discuss the specification. In the definitions below, we use *tr* to denote a transaction request and *trs* a sequence of such requests; *ai* will denote a string consisting of account information in the format used by processTrans() for outputting, and *ais* will denote a sequence of such strings. τ, as usual, will denote the trace of hook-method calls:

IsTransReq(*tr*): This predicate is *true* if *tr* is a 'legitimate' transaction request, i.e., specifies a transaction (deposit, withdraw, or printInfo), and if the transaction is deposit or withdraw, specifies a positive amount.

IsTransReqSeq(*trs*): *true* if *trs* is a sequence of legitimate transaction requests.

original model are concerned, the new model agrees with the original model.

[8] We have ignored invariants in this argument but they can be added in a straightforward manner.

$|trs|$: The length of, i.e. the number of, requests in *trs*.

$trs[j]$: The j^{th} request in *trs*.

$trs[i:j]$: The subsequence of *trs* from the i^{th} request to the j^{th}.

Trans(*tr*): The operation (deposit, withdraw, or getInfo) involved in this request; note that if the request is for printInfo, the corresponding operation is getInfo().

$trs\backslash\{\text{deposit}\}$: The subsequence of *trs* that includes only those transaction requests for which the transaction involved in the request is deposit; similarly for other transactions.

Amts(*trs*): The sequence consisting of just the *amounts* involved in the transactions in *trs* (the amount in the case of a printInfo request being taken to be 0). We will find it useful to refer to the sequence of amounts involved in, say, just the deposit transactions; this may be written as $Amts(trs\backslash\{\text{deposit}\})$.

$IRNo(trs, k)$: This value is k' if $Trans(trs[k'])$ is getInfo, and $|trs[1:k']\backslash\{\text{getInfo}\}|$ is k; in other words, $trs[k']$ is a printInfo request and is the k^{th} such request.

AccInfo(*ai*): This predicate is *true* if *ai* is a legitimate account-information string; i.e., it consists of the character "<", followed by the balance in the account, additional information (this will depend on how getInfo() is redefined in the derived class), and finally ">".

AccInfoSeq(*ais*): This predicate is *true* if *ais* is a sequence of legitimate account-information strings.

$ais[k]$: The k^{th} account-information string in *ais*.

Balance(*ai*): The balance information in the account-info string *ai*.

Info(*ai*): The *entire* information, including balance, in the account-info string *ai*. In the case of the base class, this will be identical to *Balance*(*ai*).

We should also note that when discussing our reasoning system in the preceding sections, we did not consider the case of a hook method such as getInfo() returning an explicit *result*. The record, in the trace, of a call to such a method will have to include the result returned by the method. If the k^{th} element of τ records such a call/return, we will use the notation $\tau[k].re$ to refer to the result returned by this call.

In the specification (10), we have numbered some of the lines individually as (10.1), (10.2), etc., for easy reference in the discussion. The pre-condition asserts that the hook method call trace is empty, as is results, and that transs is a legitimate sequence of transaction requests. Let us now consider the e-post-condition. (10.1) asserts that the value of transs is empty, i.e., when processTransSeq() finishes, all the transaction requests have been processed. (10.2) asserts that the final balance in the account is equal to the starting balance, plus the amounts deposited into the account, less the sum of the amounts withdrawn, in the various transactions. This follows from the fact that the starting balance in the account is balance*@pre*, and from the fact that when processing a deposit/ withdraw/ printInfo transaction, processTransSeq() invokes the deposit()/ withdraw()/ getInfo() method which means, given the specification (9) that the deposit() and withdraw() methods update the balance by the amount deposited or withdrawn and getInfo() leaves the balance unchanged, that the final balance will be as specified in (10.2).

$$epre.\mathsf{Account.processTransSeq(transss,\ results)} \equiv \quad (10)$$
$$[(\tau = \varepsilon) \wedge (\mathsf{results} = \varepsilon) \wedge (IsTransReqSeq(\mathsf{transss}))]$$
$$epost.\mathsf{Account.processTransSeq(transss,\ results)} \equiv$$
$$[(\mathsf{transss} = \varepsilon) \wedge \quad (10.1)$$
$$(\mathsf{balance} = (\mathsf{balance}@pre + \textstyle\sum Amts(\mathsf{transss}@pre \backslash \{\mathsf{deposit}\}) - \textstyle\sum Amts(\mathsf{transss}@pre \backslash \{\mathsf{withdraw}\}))) \wedge \quad (10.2)$$
$$(|\tau| = |\mathsf{transss}@pre|) \wedge$$
$$(\forall k : (1 \leq k \leq |\tau|) :: (\tau[k].hm = Trans(\mathsf{transss}@pre[k]))) \wedge \quad (10.3)$$
$$(|\mathsf{results}| = |\mathsf{transss}@pre \backslash \{\mathsf{printInfo}\}|) \wedge AccInfoSeq(\mathsf{results}) \wedge \quad (10.4)$$
$$(\forall k : (1 \leq k \leq |\mathsf{results}|) : (k' = IRNo(\mathsf{transss}@pre, k)) ::$$
$$(Info(\mathsf{results}[k]) = \tau[k'].re) \wedge \quad (10.5)$$
$$(Balance(\mathsf{results}[k]) = (\mathsf{balance}@pre + \textstyle\sum Amts(\mathsf{transss}@pre[1 : k' - 1] \backslash \{\mathsf{deposit}\}) - \textstyle\sum Amts(\mathsf{transss}@pre[1 : k' - 1] \backslash \{\mathsf{withdraw}\})))) \quad (10.6)$$
$$]$$

Figure 4: Specification of Account.processTransSeq()

The next few clauses concern the trace; they assert that the length of τ, i.e., the number of hook-method-calls is equal to the number of transactions requested; and that the particular hook method called ($\tau[k].hm$) is the one appropriate for the transaction. Note, however, that no information is provided about the *value* of the argument passed in the hook method calls (in the calls to deposit() and withdraw()). This information could have been provided by including the clause:

$$((\tau[k].hm = \mathsf{deposit}) \vee (\tau[k].hm = \mathsf{withdraw})) \Rightarrow (\tau[k].ia = Amts(\mathsf{transss}@pre)[k])$$

This simply asserts that if the hook method whose call is recorded in the k^{th} element of τ is deposit() or withdraw(), the value of the argument passed to the method is the same as the value supplied in the corresponding element of the (initial) sequence of transaction requests. In addition, information about the state (the value of balance) at the time of these calls is also not provided in (10); again, this information could have been provided with a similar clause. The fact that these items of information about these hook method calls are not included means that the base class designer does not expect enrichments that would depend on the values of the arguments passed to the hook methods or on the (base class) state at the time of the calls to the hook methods.

The remaining clauses of (10) give us information about the output that processTransSeq() will produce. (10.4) says that there will be as many elements in the final value of results as the number of printInfo transaction requests, and that each of these elements will be an account-information string. The remaining clauses are concerned with the individual elements of results; since the k^{th} element of results depends upon the portion of transss$@pre$ that precedes the k^{th} printInfo request in transss$@pre$, i.e., on transss$@pre[1 : (IRNo(\mathsf{transss}@pre, k) - 1)]$, we have introduced k' as an abbreviation for $IRNo(\mathsf{transss}@pre, k)$. (10.5) asserts that the information in this element of results is equal to the result returned by the corresponding call to getInfo() recorded in τ; this clause is important since it will allow the derived class designer to establish the enriched behavior of processTransSeq() in the derived class, in particular the enriched output that will result from a redefinition of getInfo(). The final clause (10.6) asserts that the balance information in the elements of results correspond to the actual balance in the account at the time that the information was added to results.

Showing that the body of processTransSeq() satisfies this specification is, of course, more involved than showing that methods like deposit() satisfy their specifications. This is partly due to the fact that we have to reason about the trace and partly due to the complexity of processTransSeq(). Thus,

for example, dealing with the loop in processTransSeq() would require us to introduce a suitable invariant (which would be very similar to the e-post-condition). We leave the formal statement of the loop invariant and the derivation of the e-post-condition to the interested reader.

The f-specification of processTransSeq() is easily stated:

$$pre.\text{Account.processTransSeq(transss, results)} \equiv \quad (11)$$
$$[(\text{results} = \varepsilon) \wedge (IsTransReqSeq(\text{transss}))]$$
$$post.\text{Account.processTransSeq(transss, results)} \equiv$$
$$[(\text{transss} = \varepsilon) \wedge \quad (11.1)$$
$$(|\text{results}| = |\text{transss}@pre \backslash \{\text{printInfo}\}|) \wedge AccInfoSeq(\text{results}) \wedge \quad (11.2)$$
$$(\forall k : (1 \leq k \leq |\text{results}|) : (k' = IRNo(\text{transss}@pre, k)) ::$$
$$(Balance(\text{results}[k]) = (\text{balance}@pre + \textstyle\sum Amts(\text{transss}@pre[1 : k' - 1] \backslash \{\text{deposit}\}) -$$
$$\textstyle\sum Amts(\text{transss}@pre[1 : k' - 1] \backslash \{\text{withdraw}\}))) \wedge \quad (11.3)$$
$$(\text{balance} = (\text{balance}@pre + \textstyle\sum Amts(\text{transss}@pre \backslash \{\text{deposit}\}) -$$
$$\textstyle\sum Amts(\text{transss}@pre \backslash \{\text{withdraw}\})))] \quad (11.4)$$

This states that the balance is appropriately updated corresponding to the transactions specified in transss@pre and the balance values in the results recorded in results represent the balance in the account following the completion of all earlier transactions. It is straightforward to check that the required relation, (2), between the e- and f-specifications is satisfied.

Now consider a derived class. TCAccount in Figure 5 enriches the behavior of the Account class by maintaining a count of the transactions, i.e., the number of deposits and withdrawals made on the account; the count is maintained in the variable tCount. tCount is initialized to 0 in

```
class TCAccount extends Account {
    protected int tCount;
        // current transaction count
    TCAccount(int b) { tCount := 0; }
    public void deposit(int amt)
        { balance := balance + amt; tCount++;}
    public void withdraw(int amt)
        { balance := balance − amt; tCount++;}
    public string getInfo()
        {res := string(balance); res += "trans count: "; res += string(tCount); return(res);}
}
```

Figure 5: Derived class TCAccount

the constructor. deposit() and withdraw() have been redefined to increment tCount in addition to updating balance appropriately. getInfo() has been redefined so that the result it returns contains not only the balance in the account, but also the tCount.

The specifications for these redefined methods appear in Figure 6. These are similar to those in Figure 3; the only changes are that in the post-conditions of deposit() and withdraw(), we specify how they increment tCount, and in the post-condition of getInfo(), we specify that the result returned consists of the (string representation of the) balance in the account, followed by the string "trans count: ", followed by (the string representation of the) transaction count; note that "^" in the last line of (12) denotes string concatenation. It is easy to check that the methods defined in Figure 5 satisfy these specifications and to check that the specifications in (12) and (9) meet the behavioral subclassing requirements since the pre-conditions in (12) are identical to those in (9)

$$
\begin{aligned}
&\mathit{pre}.\mathsf{TCAccount(b)} \equiv (\mathsf{b} > 0)\\
&\mathit{post}.\mathsf{TCAccount(b)} \equiv ((\mathsf{balance} = b) \wedge (\mathsf{tCount} = 0))\\
&\mathit{pre}.\mathsf{TCAccount.deposit(x)} \equiv (\mathsf{x} > 0)\\
&\mathit{post}.\mathsf{TCAccount.deposit(x)} \equiv ((\mathsf{x} = \mathsf{x}@\mathit{pre}) \wedge (\mathsf{balance} = \mathsf{balance}@\mathit{pre} + \mathsf{x}) \wedge\\
&\qquad\qquad (\mathsf{tCount} = \mathsf{tCount}@\mathit{pre} + 1))\\
&\mathit{pre}.\mathsf{TCAccount.withdraw(x)} \equiv (\mathsf{x} > 0)\\
&\mathit{post}.\mathsf{TCAccount.withdraw(x)} \equiv ((\mathsf{x} = \mathsf{x}@\mathit{pre}) \wedge (\mathsf{balance} = \mathsf{balance}@\mathit{pre} - \mathsf{x}) \wedge\\
&\qquad\qquad (\mathsf{tCount} = \mathsf{tCount}@\mathit{pre} + 1))\\
&\mathit{pre}.\mathsf{TCAccount.getInfo()} \equiv (\mathit{true})\\
&\mathit{post}.\mathsf{TCAccount.getInfo()} \equiv ((\mathsf{balance} = \mathsf{balance}@\mathit{pre}) \wedge (\mathsf{tCount} = \mathsf{tCount}@\mathit{pre}) \wedge\\
&\qquad\qquad (\mathit{result} = (\mathit{string}(\mathsf{balance}) \hat{} \text{“trans count: ”} \hat{} \mathit{string}(\mathsf{tCount})))) \qquad (12)
\end{aligned}
$$

Figure 6: Specification of TCAccount class

and the post-conditions in (12) imply the corresponding post-conditions in (9)[9].

Let us now turn to the essential point of our reasoning task, that of incrementally arriving at the richer behavior of TCAccount.processTransSeq() due to the richer behavior of the methods it invokes. The key clause in the base class (e-)specification of processTransSeq() that allows such enrichment is (10.5):

$$(\mathit{Info}(\mathsf{results}[k]) = \tau[k'].re)$$

First recall, according to the relation between k and k' in (10), that $\tau[k']$ records the k^{th} call to getInfo(). Now, *post.*TCAccount.getInfo() specified in (12) gives us more information about the result returned by this call, in other words about the value of $\tau[k'].re$, than does *post.*Account.getInfo() specified in (9). Specifically, whereas (9) states that the result returned by getInfo() will include the string representation of the balance in the account as a prefix, (12) states what the rest of the result returned by (TCAccount.)getInfo() consists of: the string "trans count:" followed by the string representation of the value of tCount in the account.

What will this value be? According to the specification (12), TCAccount.deposit() and TCAccount.withdraw() both increment tCount by 1. So the value that (TCAccount.)getInfo() reports for tCount in the result it returns will depend on the number of calls made so far to these methods. And since these are all hook methods, calls to these methods are all recorded on τ. We first introduce a couple of additional auxiliary functions and predicates which will be of use in stating the richer behavior of processTransSeq():

TCAccInfo(*ai*): This predicate is *true* if *ai* is a legitimate TCAccount-information string; i.e., it consists of "<", the balance in the account, the string "trans count: ", an integer (being the value of the transaction count in the account), and finally, ">".

TCAccInfoSeq(*ais*): This is *true* if *ais* is a sequence of legitimate TCAccount-information strings.

TransCount(*tcai*): The trans count value recorded in the TCAccount-information string *ai*.

The specification of TCAccount.processTransSeq() appears in Figure 7. The pre-condition is the same as in the base class-specification. (13.1) is simply the post-condition from the base class-specification; (13.2) and (13.3) specify the enrichment. (13.2) should be compared with the (second

[9]The invariants for both Account and TCAccount are *true*.

$$
\begin{aligned}
&\mathit{epre}.\mathsf{TCAccount.processTransSeq(transs,\ results)} \equiv \\
&\qquad [(\tau = \varepsilon) \wedge (\mathsf{results} = \varepsilon) \wedge (\mathit{IsTransReqSeq}(\mathsf{transs}))] && (13)\\
&\mathit{epost}.\mathsf{TCAccount.processTransSeq(transs,\ results)} \equiv \\
&\quad [\ \mathit{post}.\mathsf{Account.processTransSeq(transs,\ results)} \wedge && (13.1)\\
&\quad \mathit{TCAccInfoSeq}(\mathsf{results}) \wedge && (13.2)\\
&\quad (\forall k : (1 \leq k \leq |\mathsf{results}|) : (k' = \mathit{IRNo}(\mathsf{transs}@\mathit{pre}, k)) :: \\
&\qquad (\mathit{TransCount}(\mathsf{results}[k]) = \\
&\qquad\quad (\mathsf{tCount}@\mathit{pre} + |\mathsf{transs}@\mathit{pre}[1 : k' - 1] \backslash \{\mathsf{deposit},\ \mathsf{withdraw}\}|))) && (13.3)\\
&\quad]
\end{aligned}
$$

Figure 7: Specification of TCAccount.processTransSeq()

conjunct of) (10.4); whereas the latter tells us that results is a sequence of strings each of which consists of "<", followed by the balance in the account, followed (possibly) by some additional information, followed by ">", (13.2) also tells us that this additional information will be the string "trans count:" followed by (the string representation of) the value of tCount in the account at the time this was added to results. And (13.3) tells us that this value will be equal to the value of tCount at the start of TCAccount.processTransSeq(), plus the number of deposit and withdraw transaction requests preceding the printInfo request that led to this TCAccount-information string being added to results.

Let us now see how we can, by using our Enrichment Rule **R2**, establish (13), given the base class e-specification (10) and the derived class behaviors of the hook methods specified in (12). The first antecedent of **R2** is immediate since the e-pre-conditions for the base and derived classes are identical. Now consider the second antecedent. Note first that the various clauses ((10.1) through (10.6)) in *epost*.Account.processTransSeq(transs, results) are such that the substitutions $-\tau$ by $\tau \downarrow b$, etc.– specified in the first clause of the left side of this antecedent have no effect since balance is a member of the base class (hence also of the derived class), transs and results are arguments of the method (hence are the same in the base and derived classes), and $\tau[k].hm$ (the identity of the hook method invoked in the k^{th} element of τ) and $\tau[k].re$ (the result returned by this call) are the same in the base and derived classes. Therefore, (13.1) will be satisfied (given the first clause of the left side of the second antecedent of **R2**). (13.2) may be established as follows: From the second clause of (10.4) we know that each element of results is an account-information string; and from (10.5) we know that the information in the k^{th} element of results is the same as the result returned by k^{th} call to getInfo(); (13.2) then follows from what *post*.TCAccount.getInfo() (defined in (12)) tells us about the result returned by TCAccount.getInfo().

Next consider (13.3). Appealing again to (10.5), we can conclude that $\mathit{TransCount}(\mathsf{results}[k])$ is equal to $\mathit{TransCount}(\tau[k'].re)$ where k' is $\mathit{IRNo}(\mathsf{transs}@\mathit{pre}, k)$. The specification (12) of TCAccount.getInfo() tells us that the transaction count in the result returned by this method is the same as the value of tCount at the time the method was called; i.e., equal to the value of tCount in the state $\tau[k'].is$. The clause *ncbc*() in the left side of the second antecedent of **R2** tells us, given that tCount is introduced in the derived class TCAccount, that the value of this variable as recorded in the "initial state" (the *.is* component) of each element of τ is the same as its value in the "final state" (the *.fs* component) of the previous element; the clause *ccds*() in the same antecedent tells us that the states as recorded in the *.is* and *.fs* components of each element of τ satisfy the post-condition, specified in (12), of the corresponding hook method. Since, according to (12), TCAccount.deposit() and TCAccount.withdraw() each increment tCount by 1, and TCAc-

count.getInfo() leaves it unchanged, we can then conclude that the transaction count in the result returned by the call to getInfo() recorded in the k'^{th} element of τ is equal to the value of tCount at the start of processTransSeq() plus the number of calls to deposit()/withdraw() recorded in the first $(k'-1)$ elements of τ. This, combined with (10.3), lets us conclude that (13.3) must be satisfied.

And finally, consider the f-specification of TCAccount.processTransSeq(); we express this in terms of the pre- and post-conditions of the f-specification of Account.processTransSeq(), spelling out only the additional clauses:

$$pre.\mathsf{TCAccount.processTransSeq(transss, results)} \equiv \quad (14)$$
$$pre.\mathsf{Account.processTransSeq(transss, results)}$$

$$post.\mathsf{TCAccount.processTransSeq(transss, results)} \equiv$$
$$[post.\mathsf{Account.processTransSeq(transss, results)} \wedge \quad (14.1)$$
$$TCAccInfoSeq(\mathsf{results}) \wedge \quad (14.2)$$
$$(\forall k : (1 \leq k \leq |\mathsf{results}|) : (k' = IRNo(\mathsf{transss}@pre, k)) ::$$
$$(\, TransCount(\mathsf{results}[k]) =$$
$$(\mathsf{tCount}@pre + |\mathsf{transss}@pre[1\!:\!k'-1] \backslash \{\mathsf{deposit}, \mathsf{withdraw}\}|)) \,)\,] \quad (14.3)$$

Again it is straightforward to check that the relation (2) holds between the e-specification (13) and the f-specification.

Note that the clause (14.1) follows from behavioral subclassing considerations, given that (12) is consistent with (9) If all we were interested in was to show that TCAccount.processTransSeq() behaves in a way consistent with the (f-)specification of Account.processTransSeq(), we would not need the formalism developed in this paper. But, of course, the whole point of defining the derived class, in particular of redefining the hook methods in the TCAccount class, was to *enrich* the behavior of the template method processTransSeq() as specified in (14.2) and (14.3). And it is this enriched behavior that our formalism allows us to establish. And in establishing this enriched behavior, we did not have to reanalyze the behavior of the code of this method; instead, we plugged in the richer behavior of the derived class hook methods into the e-specification, established during the base-class analysis, of the template method.

We will conclude this section with two remarks. First, suppose we defined a variation of TCAccount in which only *large* transactions, i.e., those in which the amount involved is greater than 5000 are counted. Then we cannot reason about the resulting richer behavior of processTransSeq() on the basis of the specification (10) since that specification does not tell us what argument values processTransSeq() passes to the hook methods it calls. It would have been easy enough to include this information in (10); it is upto the base class designer to anticipate what kinds of enrichments might be implemented in the derived classes and include the appropriate information in the e-specifications. Being too liberal here, that is allowing for all kinds of enrichments, would lead to very complex e-specifications; being too conservative will make it impossible to reason incrementally about enrichments that were not anticipated. This is a trade-off between flexibility of design versus complexity of specs.

Our second remark has to do with the nature of our e-specifications. E-specifications are most conveniently expressed, as in the case of Account.processTransSeq(), by first defining some useful functions on traces that, in a sense, mimic the behavioral pattern exhibited by the template method, and then writing down the e-specification of the method in terms of these functions. For more complex situations, we believe it would be useful to introduce specialized notation for use in writing such specifications, perhaps using constructs similar to those of regular expressions; we plan to investigate such notations in future work.

6 Related Work

A number of authors have addressed questions relating to reasoning about behavioral issues in OO systems. Lamping [Lam93] proposes specifying, for each polymorphic function of a class, the set of virtual functions that it invokes. This will allow a derived class designer to know whether a given polymorphic function might be affected –enriched in our terminology– by redefinitions of specific virtual functions. The idea seems to be that the designer can then go back and study the code of the polymorphic function in the base class to see how it is affected; our goal of course is to try to avoid such reanalysis. Kiczales and Lamping [KL92] propose providing information not just about which virtual functions the polymorphic function will call but also the order in which it will call them. But they don't talk about establishing behavioral specifications or about arriving at the enriched behavior of the polymorphic method in the derived class by plugging in, into the base class specification, information about the behavior of the redefined virtual methods.

Behavioral problems arising from careless use of inheritance have been discussed by a number of authors, see for example [Sak89]. It is, as we noted earlier, to address this problem that the notion of behavioral subtyping [LW93, LW94] was developed; our work extends this since our goal is not just to guarantee that the base-class-level analysis of the template method remains valid in the derived class but also to reason about the richer behavior of the template method in the derived class. We should also note that a class A may be a behavioral subtype of a class B independently of whether or not A is defined as a derived class of B. Dhara and Leavens [DL96] focus on the conditions that will ensure that a derived class will be a behavioral subtype of its base class so there is a natural connection to our work since the primary focus of this paper is the relation between the behaviors of derived and base class. But note that Dhara and Leavens, like other authors who deal with behavioral subtyping, do not address the question of the enriched behavior resulting from the redefinition of methods in the derived class which is our main concern. Stata and Guttag's [SG95] interest is somewhat similar to that of [DL96]. They extend the notion of behavioral subtyping to deal with redefinitions of *groups* of virtual methods in the derived class, but again the question of reasoning about the richer behavior in the derived class is not addressed. Edwards [Edw97] considers the reasoning reuse that may be achieved if the derived class is not necessarily a behavioral subtype of the base class but certain other conditions, such as the invariant for the derived class being the same as that for the base class, are satisfied. But as we just saw, if we want to be able to reason incrementally about the behavior of template methods, behavioral subtyping (or rather behavioral subclassing) is essential.

Abadi and Leino [AL97] propose a logic for reasoning about OO programs expressed in a simple language that they define. They do not have classes in their language; each object, in the logic, 'carries' with it the specifications of its various methods. In addition, the logic takes explicit account of object creation via the *alloc*() function; this allows them to deal with aliasing between objects. But [AL97] does not address the question of reasoning about polymorphic methods; in particular it is not clear that we would be able to reason incrementally about the behavior of the polymorphic method from its specification in the base class (or object); instead, it seems likely that one would have to re-reason about the (body of the) polymorphic method in the context of the new class to arrive at its derived-class behavior.

Buchi and Weck's [BW99] work is closer to our approach. They note that pre- and post-conditions on just the values of member variables are inadequate when dealing with template methods and that one must also make use of traces. They introduce a formalism and a programming language-like notation using which some information about the trace of hook method calls can be

specified. Although their use of traces is similar to ours, Buchi and Weck focus only on specifying conditions that the trace must satisfy, not the question of how to use such specifications to arrive at the richer behavior that results from the redefinitions of the hook methods in the application. It is also worth noting that traces have been used extensively [Dah92, Hoa85, MC81] for reasoning about communicating processes. The soundness and completeness arguments we sketched are quite similar to the proofs of soundness and completeness of a trace-based CSP proof system in [SD82].

Keidar *et al.* [KKLS00] present a formal system for arriving at specifications and proofs incrementally. But the kind of inheritance they use is not related to inheritance (of code) from base to derived classes in standard OO languages; rather, their 'inheritance' has to do with starting with the specification for an automaton and arriving at the specification for another automaton that exhibits additional behavior. In particular, [KKLS00] does not deal with incremental reasoning about the behavior of template methods.

Garlan *et al.* [GJND98] develop a temporal-logic based approach to reasoning about *implicit invocation*. Calls to hook methods from template methods can be considered implicit invocations since the actual method invoked cannot be determined from just the body of the template method but also depends on the derived class under consideration. While there are some similarities with our work, a key difference is that whereas we first reason about the base class and then arrive incrementally at the behavior of the derived class, [GJND98] takes a very different approach: Given a system S (consisting of all the methods defined in *all* the classes) and a specification for S, partition S into a number of groups, arrive at a suitable specification for each group, show that each group satisfies its specification, and show that together the specifications of the individual groups imply the original specification of S.

7 Discussion

One of the most important ideas introduced by *Simula* was the notion of polymorphism. Polymorphism allows a derived class designer to enrich the behavior of the template methods of the base class by redefining one or more of the hook methods that the template methods invoke. If the base class has been designed carefully and includes the right hooks and the template methods invoke these at the right points, different derived class designers can achieve different enrichments, appropriate to their particular applications, with relatively little effort; much has been written about the central role that polymorphism plays in building flexible, extensible OO systems see, for example, [Mey97]. The work reported in this paper has been motivated by the belief that the techniques that we use to reason about the behaviors exhibited by such systems must similarly be incremental, in other words, that we must provide suitable characterizations of the behaviors of the base class template methods so that we can arrive at the richer behaviors they exhibit in the derived class by simply plugging-in appropriate information about the redefined hook methods. Although much work has been done in the past few years in developing reasoning systems for dealing with OO systems, most of this work, in particular the work on behavioral subtyping (and subclassing), has focused on ensuring that base class specifications of template methods continue to be satisfied even with the derived class redefinitions of the hook methods. Our main contribution has been to extend this to allow us to reason also about the richer behavior that the template methods exhibit as a result of these redefinitions of the hook methods.

The key component of our approach that makes it possible to reason about this richer behavior without having to reanalyze the code of the template method is what we called the e-specification

of a template method. The e-specification gives us information about the trace of hook method calls that the template method in question makes and its relation to the behavior the template method exhibits. Although the e-specification is more complex than the standard (f-)specification that only specifies information about the values of the member variables of the class, it is clear that an incremental reasoning system must include information about the trace of hook method calls since that is the source of the power of polymorphism.

In this paper, we have considered only a single base class and the derived classes that might be defined from it. In a complex OO system, there will of course be many objects that are instances of a variety of classes. Indeed, this corresponds naturally to a distributed system with the individual objects corresponding to the processes of the distributed system. There are of course additional issues to be considered in such a system such as synchronization. It has been observed that this can introduce additional new problems such as the *inheritance anomaly* [MY93]. In future work, we hope to extend our reasoning system to deal with behavioral issues in such systems. The fact that traces which play such a key role in our system also occur naturally when dealing with the behavior of distributed systems [Hoa85, MC81] suggests that such an approach is indeed reasonable.

References

[AL97] M. Abadi and K. Leino. A logic of oo programs. In *Proceedings of TAPSOFT '97*, pages 682–696. Springer-Verlag, 1997.

[Ame91] 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.

[BW99] M. Buchi and W. Weck. The greybox approach: when blackbox specifications hide too much. Technical Report TUCS TR No. 297, Turku Centre for Computer Science, 1999. available at http://www.tucs.abo.fi/.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 1985.

[Dah92] O.-J. Dahl. *Verifiable Programming*. Prentice-Hall, 1992.

[DL95] K.K. Dhara and G.T. Leavens. Weak behavioral subtyping for types with mutable objects. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Proc. of 11th Annual Conf. on Math. Found. of Programming*, Elec Notes in Theoretical Computer Sc., pages 269–290. Elsevier, 1995.

[DL96] K.K. Dhara and G.T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proc. of 18th Int. Conf. on Softw. Eng.*, pages 258–267. IEEE Computer Soc., 1996.

[DN66] O.-J. Dahl and K. Nygaard. SIMULA - An Algol-based Simulation Language. *Communications of the ACM*, 9(9):671–678, Sept 1966.

[Edw97] S. Edwards. Representation inheritance: A safe form of 'white box' code inheritance. *IEEE TSE*, 23, 83-92, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable OO Software*. Addison-Wesley, 1995.

[GJND98] D. Garlan, S. Jha, D. Notkin, and J. Dingel. Reasoning about implicit invocation. In *Proceedings of Foundations of Software Engineering (FSE-6)*, pages 209–221. ACM Press, 1998.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Jon90] C. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1990.

[KKLS00] I. Keidar, R. Khazan, N. Lynch, and A. Shvartsman. Inheritance-based technique for building simulation proofs incrementally. In M. Harrold, editor, *22nd Int. Conf. of Software Eng.*, pages 478–487. ACM, 2000.

[KL92] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. In *OOPSLA '92*, pages 435–451, 1992.

[Lam93] J. Lamping. Typing the specialization interface. In *OOPSLA*, pages 201–214, 1993.

[LW93] B. Liskov and J. Wing. A new definition of the subtype relation. In *ECOOP*, 1993.

[LW94] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. on Prog. Lang. and Systems*, 16:1811–1841, 1994.

[MC81] J. Misra and K. Chandy. Proofs of networks of processes. *IEEE Trans. on Software Eng.*, 7:417–426, 1981.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[MY93] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in oo concurrent languages. In Agha, Wegner, and Yonezawa, editors, *Research directions in concurrent OO programming*, pages 107–150. MIT Press, 1993.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(1):319–340, 1976.

[Sak89] M. Sakkinen. Disciplined inheritance. In S. Cook, editor, *Proceedings of ECOOP '89*, pages 39–56. British Computer Workshop Series, 1989.

[SD82] N. Soundarajan and O.-J. Dahl. Partial correctness semantics of CSP. Technical Report 66, Institute of Informatics, Oslo University, 1982.

[SG95] R. Stata and J.V. Guttag. Modular reasoning in the presence of subclassing. In *OOPSLA*, pages 200–214. ACM Press, 1995.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Langauge*. Addison-Wesley, 1999.