Framework-Based Applications: From Incremental Development to Incremental Reasoning

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

Abstract

Object-Oriented frameworks provide a powerful technique for developing groups of related applications. The framework designer designs the main classes and methods, including the *template* methods that direct the 'flow-of-control' and at appropriate points call various *hook* methods. The application developer provides suitable definitions for the hook methods; a different developer could produce a different application by providing an alternate set of definitions for these methods. Our goal is to develop an *incremental* technique for reasoning about the framework and the applications. Using the proposed technique, we would reason about the framework behavior just once; whenever a new application is developed using the framework, we would incrementally arrive at its behavior by reasoning about the behavior of the newly defined hook methods, and then 'plugging-in' this behavior into the behavior of the framework.

We illustrate the technique by applying it to a simple example. And we contrast our approach with that based on the notion of *behavioral subtyping*.

1 Introduction

Object-Oriented frameworks [3, 10] provide a powerful technique for developing groups of similar or related applications. The framework designer who should have a deep understanding of the application area, designs the main classes and methods, including in particular the key methods that direct the 'flow-of-control', and call appropriate methods of various classes. One or more of the called methods will be *virtual*[1] or pure virtual; indeed, one or more of the classes may just be *interfaces* (*abstract base classes* in *C++* terminology), i.e., consist entirely of pure virtual methods. In the patterns literature [4], the method(s) directing the flow-of-control are called *template* methods, and the (pure) virtual methods that they invoke, *hook* methods. In order to implement a complete application based on this framework, the application developer would just have to provide suitable definitions for all the hook methods; a different developer could produce a (possibly very) different application by providing an alternate set of definitions for the hook methods. If the framework has been designed carefully and provides the right hooks, an entire new application can be developed with just the incremental effort involved in designing a new set of hook methods.

Our goal in this paper is to develop a similarly incremental technique for reasoning about the framework and the applications. Using our technique, we would reason about the framework behavior just once; every time a new application is developed using this framework, we would incrementally arrive at the behavior of this application by first reasoning about the behavior of the newly defined hook methods, and then 'plugging-in' this behavior into the behavior of the framework. The standard approach to dealing with applications based on OO frameworks is based on the notion of *behavioral subtyping* [6, 7, 8]. While behavioral subtyping is important, it focuses on behavior that is common to all the applications built on a given framework, completely ignoring application-specific behavior. The approach proposed in this paper builds on behavioral subtyping and makes it possible to specify the common behaviors, as well as application-specific behaviors.

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

- Makes the case that behavioral subtyping by itself is not sufficient to deal with framework based applications since it is concerned only with behavior that is common to all the applications based on the given framework.

[1] For concreteness we often use *C++/Java*-terminology, but our approach is not language specific.

- Presents a specification notation and reasoning technique using which we can include sufficient information in the specifications of template methods, so that information about the behavior of hook methods –as defined in the application– can be plugged-in, to arrive at specifications appropriate to the particular application.
- Illustrates the approach by applying it to a simple example.

The rest of the paper is organized as follows: In Section 2, we briefly summarize the behavioral subtyping based approach to dealing with framework-based applications. In Section 3 we consider, via a simple framework and several applications that might be built on it, what additional information needs to be included in our specifications of template methods, and how this may be done. In Section 4 we see how application-specific information about the hook methods may be plugged into the specification of the template methods. The presentation in these two sections omits some of the formal detail since our focus is on motivitating and intuitively justifying the approach rather than on formalism; formal details including questions of soundness will be presented in a separate paper. In Section 5 we apply the approach to a simple example, a continuation of the one used in Section 3. In the final section we reiterate the need for going beyond behavioral subtyping in dealing with frameworks, summarize how our approach does so, and point out problems that remain to be addressed.

2 Behavioral Subtyping

Let us consider an extremely simple situation in which the framework consists of a single class C that has one non-virtual method $t()$ and a single virtual[2] method $h()$; C will also typically include one or more data members. In the design patterns terminology, $t()$ is the template method of this 'framework' and it invokes (one or more times) the hook method $h()$. In order to develop an 'application' using this framework, the application developer would define a derived class D of C that would provide a definition for $h()$. A client programmer who wishes to use the application would create an object X of type D and invoke the method $t()$ on it. OO polymorphism will then ensure that during this execution of $t()$, the invocations of $h()$ will in fact call the method defined in D, so that this call to $t()$ will exhibit the behavior appropriate for this particular application.

[2]In *Java* and *Eiffel* all methods are virtual by default, and the non-virtual methods, that is those that are not to be redefined in the application, are flagged as *final*; in *C++*, final is the default, and *virtual* methods are so flagged.

The standard approach, based on the notion of behavioral subtyping [6, 7, 8], to specifying such a framework may be summarized as follows:

> First arrive at a specification S_t for the template method $t()$; this specification consisting in the usual manner of pre- and post-conditions, will give us the effect that an execution of the method has on the values of member variables of C. Also provide a specification S_h for the hook method $h()$, specifying its effect on the values of the member variables of C. Next, validate the specification S_t –either formally or informally– by analyzing the body of $t()$ as given by its definition in the class C; during this analysis, make use of the specification S_h for analyzing the effects of the calls to $h()$ that appear in $t()$. If C does not provide a definition for $h()$ (so $h()$ is a pure virtual method in C), then we are done with reasoning about C. If C does include a definition for $h()$ (so the 'framework' C is, by itself, a complete application), then ensure that this definition satisfies the specification S_h.
>
> Next, when we develop an application by defining a derived class D of C and providing a definition in D for the method $h()$, arrive at an appropriate specification S'_h corresponding to this $h()$, check that the definition of $h()$ in D satisfies S'_h, *and* show that S'_h is *consistent* with S_h; more precisely, show that any method that satisfies S'_h must necessarily also satisfy S_h.

It is that last step of showing that the specification S'_h of $h()$ corresponding to its definition in D is consistent with the specification S_h in C that is the crucial to the approach. It ensures that D is a *behavioral subtype* of C, in other words that the behavior that methods of D exhibit is consistent with the behavior of the methods of C as specified by S_t and S_h. We do not need to reanalyze $t()$ to ensure that it will satisfy S_t even though the calls to $h()$ that this $t()$ makes will be to the $h()$ defined in D, rather than the one defined in C. This is because in our previous analysis of $t()$ to check that it satisfies S_t, the only assumption we made concerning the effect of the calls to $h()$ was that they would satisfy the specification S_h. Now although we have provided a new definition for $h()$ in D, this new definition has been shown to satisfy the specification S_h, hence $t()$ will –assuming that our previous analysis was correct– continue to satisfy S_t.

It would seem from the above discussion that this approach satisfies the requirement of incrementality in reasoning about framework-based applications: We first reasoned about the framework (the specifications S_t and S_h); then when we defined a new application D, all we needed to do was the incremental effort of checking that the new definition of $h()$ satisfies its specification S'_h and that hence it would satisfy its original specification S_h. But this ignores an important point: The power of the framework approach derives from the fact that the application developer can, when he[3] defines the hook method $h()$ in the derived class D, tailor this definition to suit the needs of this particular application; OO polymorphism ensures that although the definition of $t()$ is inherited from the framework, that as far as clients of this application are concerned, the $h()$ that will be invoked during the execution of $t()$ will be the one defined in D; hence the method $t()$ will exhibit the behavior tailored to this application. But the specification S_t is *not* so tailored; rather, it is written so that they would be valid for *all* applications that are built on this framework. In other words, as far as the behavioral subtyping approach is concerned, there is no difference between the different applications that may be built on this framework!

What we need then is a specification of $t()$ that can be tailored to the individual D by 'plugging-in' the specification of $h()$ corresponding to that particular D. Such a specification will capture the unique (as well as the common) behavior provided by that application. This is very important to the client because the reason he chose to use *this* application rather than one of the others is to exploit the unique behavior provided by this application. Thus a specification technique that brings out the unique behaviors in the different applications built on a common framework is key to utilizing the power of the framework approach; *'vive la difference'*, so to speak!

We conclude this section with a note about one point that we have ignored so far – abstraction. Clients of classes should not be concerned with internal details of member variables of the class, or how the various methods modify the values of these variables. The standard method of addressing this is to have two specifications, a concrete one for the class designer/implementor, and an abstract one for the client. All of this applies to the reasoning technique we develop in this paper in a standard manner and hence we will omit further discussion of it, using only concrete specifications throughout. Once the concrete specification has been obtained for an application, we can provide an abstract specification for it and relate it to the concrete specification in the standard manner.

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

3 Specifying the Framework

What extra information do we need to include in the specification of a template method $t()$ so that we can plug-in appropriate information about the hook method from the application? The example framework we consider next and the applications that may be built on it, will help us answer this question. Our example is a simple banking framework, specifically a general type of bank account. The definition[4] of the BankAccount class appears in Figure 1. Deposit and

```
class BankAccount {
  protected int bal;
      // current account balance
  protected nmonthly;
      // no. of monthly transactions
  protected Array<int> monthlies;
      // transactions which happen every month

  BankAccount(int b)
    { bal = b; monthlies.init(0); nmonthlies = 0; }
  public void Deposit(int amt)
    { bal = bal + amt; }
  public void Withdraw(int amt)
    { bal = bal − amt; }
  public final int GetBalance() { return bal; }
  public final void AddMonthly(int amt)
    { if (monthlies.size() <= nmonthlies)
        { monthlies.grow(5); }
      monthlies[nmonthlies++] = amt; }
  public final void NewMonth()
    { for (int i = 0; i < nmonthlies; i++) {
        if (monthlies[i] > 0)
          { this.Deposit(monthlies[i]); }
        else { this.Withdraw(−monthlies[i]); } }
    }
}
```

Figure 1: Class *BankAccount*

Withdraw, as their names suggest, are used to deposit money into and withdraw money from the account. It is expected that the application designer will provide alternate definitions for these two methods tailored to the kind of bank account needed for his particular application, that is why these are not flagged as final.

monthlies[] is an array of transactions that is expected to be performed once every month; new transactions may be added to this array by calling the AddMonthly method. NewMonth is the method that will perform all the transactions in this array and is expected to be invoked once a month. When NewMonth

[4]For concreteness, we use a *Java*-style syntax in examples.

is invoked, it does not directly update bal, but instead calls Deposit and Withdraw to perform the transactions; so if the application developer provides new definitions for Deposit and/or Withdraw, NewMonth will invoke these newly defined methods. Thus NewMonth is the template method of our 'framework' and Deposit and Withdraw the hook methods.

How do we specify the behavior of these methods? Consider the Deposit method:

$$\begin{aligned}&\mathit{Deposit(amt).pre} \equiv \mathit{true}\\&\mathit{Deposit(amt).post} \equiv\\&\quad (\mathit{bal} = \#\mathit{bal} + \mathit{amt})\\&\quad \wedge\ !\{\mathit{nmonthlies},\ \mathit{monthlies},\ \mathit{amt}\}\end{aligned} \tag{1}$$

As usual, pre-conditions are assertions on the state of the object and parameters immediately before the method is invoked[5]; post-conditions are assertions over the state immediately before *and* the state immediately after a method is invoked. The specification states that if the pre-condition holds for the state before a method invocation, then the post-condition is guaranteed to hold over that same state and the state when the method finishes. In our post-conditions, we use the notation *#var* to refer to the value of a variable in the state before invocation and *var* to refer to the variables value in the final state. Writing the name of a variable within braces preceded by "!" is a shorthand way of specifying that the value of that variable is the same in the initial and final states. Thus the post-condition above asserts that Deposit updates bal as expected, and does not affect the array of monthly transactions, the number of those transactions, or the amt parameter. Withdraw can be specified similarly.

What of the template method? How do we specify its behavior? A specification similar to that of Deposit would look like:

$$\begin{aligned}&\mathit{NewMonth().pre} \equiv \mathit{true}\\&\mathit{NewMonth().post} \equiv\\&\quad (\mathit{bal} = \#\mathit{bal} + \Sigma_{i=0}^{\mathit{nmonthlies}-1}\ \mathit{monthlies}[i])\\&\quad \wedge\ !\{\mathit{nmonthlies},\ \mathit{monthlies}\}\end{aligned} \tag{2}$$

This states that bal will be updated to account for all the deposits and withdrawals in monthlies[], and the other variables unaffected. But this specification will *not* be sufficient from the point of view of the application designer. It does summarize the effect of the NewMonth method on the variables of BankAccount, but it makes no mention of the fact that NewMonth calls the hook methods Deposit and Withdraw. Why is this information important? An example 'application' will tell us.

Suppose the application designer wants to create a a *promotional account*. Perhaps the bank has entered into a promotional agreement with an airline to offer frequent flier miles to the bank's customers; the more a customer uses his account, the more miles he earns. The designer can implement this as class PromoAccount that inherits from BankAccount, adds a data member miles which stores the number of miles accumulated by a customer, and methods for getting and resetting this value. Most importantly, PromoAccount would include new versions of Deposit and Withdraw which update miles appropriately. A simple scheme that awards twenty-five miles for each transaction is easily implemented, as in Figure 2.

```
public class PromoAccount extends BankAccount {
  protected int miles; // no. of miles earned
  PromoAccount(int b)
    { miles = 0; super(b); }
  public void Deposit(int amt)
    { super.Deposit(amt); miles += 25; }
  public void Withdraw(int amt)
    { super.Withdraw(amt); miles += 25; }
}
```

Figure 2: Class *PromoAccount*

PromoAccount will inherit BankAccount's facilities for monthly scripted transactions. Moreover, since NewMonth invokes the hook methods Deposit and Withdraw to perform these transactions, the frequent filer miles total for the account will be properly updated which, of course, is the whole point of the example.

But we cannnot arrive at this conclusion from the specification (2) since it does not tell us that NewMonth invokes Deposit or Withdraw, so there is no way to know what effect NewMonth may have on miles. The application designer might even conclude (incorrectly) from (2) that NewMonth will have *no* effect on miles, and provide a new body for this method in PromoAccount to ensure correct updating of miles. (This would not just lead to unnecessary work– if the new body first calls the base class version to perform the transactions, and then updates the milage total, the account would receive twice as many miles as it should ... a bug!) Hence, specifications of template methods should contain at least the following information:

- Names of the hook methods it invokes.
- Number of times it invokes each hook method.

Given this information, the designer could mentally

[5] In practice it would be a good idea to include, in the pre-condition of Deposit, a requirement that amt > 0; in the interest of keeping our specifications simple, we often ignore such details.

"plug-in" the behavior of the new versions of the hook methods to conclude that NewMonth, as defined in the framework, will update miles appropriately.

Are these two pieces of information always sufficient? Unfortunately, the answer is no. Suppose the application designer wanted to use a more complicated scheme for awarding miles; suppose, for example, that in order to encourage deposits and discourage withdrawals, the designer deems that each successive deposit would result in an award of double the number of miles awarded by the previous deposit. However, a withdrawal results in no miles being awarded and a resetting of the "deposit award" to twenty-five. Such a scheme could be easily coded, but now even if the specification of NewMonth said that Deposit and Withdraw are called once for each positive and negative element respectively in the monthlies array, the application designer would be unable to reliably conclude how many miles are awarded by a particular call to NewMonth. The problem is that, quite possibly, NewMonth might perform all the deposits before all the withdrawals (the application designer cannot rule this out based on the specification we just considered), and a much larger number of miles will be awarded than if NewMonth, as is actually the case, performs the deposits and withdrawals in the order that they appear in the monthlies array. Hence, in general, we need another piece of information to be added to the specification of a template method:

- The *interleaving* of the calls to the hook methods.

And we are still not quite finished. Still more mileage schemes can be envisioned. For example, suppose twenty-five miles are given for both deposits and withdrawals, but not if the withdrawal overdraws the account. In such a case, knowing only the interleaving of the calls to Deposit and Withdraw would not allow the designer to figure out whether or not one of the calls to Withdraw would result in an overdraft. For this, the specification of NewMonth would have to include information about the parameters of each call to the hook method (the parameters to each call are the corresponding values in the monthlies array) as well as information about the value of bal –in general the state of the object– immediately preceeding each call to a hook method. Similar arguments can be made to show that we may also need to know about the state immediately after the hook method's return. So we have a fourth kind of information that should be present in the specification of a template method:

- The state of the object and parameters immediately before and after each hook method call.

How do we include information about these four items in the specification of a template method? With each such method t, we associate a *hook method call trace*, τ. In this variable we record the interactions between the template method t and the hook methods it invokes. τ will be a *sequence*; each of its elements will represent a single call to (and return from) a hook method; these elements will be in the same order as the hook method calls. τ will have the value ϵ, the empty sequence, at the start of t (since at its start, t has not yet invoked any hook methods). The element corresponding to a call will record the name of the hook method called, the values of the parameters passed, the state (i.e., the values of the member variables) of the class at the time of the call; the values returned by the hook method, and the state of the class at the time of the return. The *specification* of t will include assertions about the form and content of τ as well as relations between the various elements of τ with each other and with the final state of the class when t finishes as well as any values t may return. In the worst case the resulting specifications can be unwieldy, but in practice for most frameworks the needed information can be expressed in a fairly concise and comprehensibe manner.

It is useful to note what is *not* included in τ: No information about the sequence of operations that the template method performs between successive calls to the hook methods is recorded; also not recorded are calls to non-hook methods that the template method might make. In other words, information about the state of the object is recorded only at those points where the template method interacts with the hook methods. This is precisely the information the application designer needs in order to arrive at the complete behavior of the template method once he has specified the behavior of the hook method bodies as defined in his specific application.

4 Incremental Reasoning

In this section we present our reasoning technique. We discuss both reasoning performed by the framework designer and that performed by the application developer who uses the framework. For his part, the framework designer must check that the pre- and post-condition specification for each method of the framework is indeed implemented correctly by the code, in the framework, for that method. For non-template methods, this is straightforward and standard techniques, be they formal or informal, may be used. For a template method $t()$, the task is a bit more complex since its specification will likely contain assertions about its hook method call trace τ; so we must check that if the pre-condition of $t()$ is satisfied when it is invoked, then the behavior of $t()$, that is the values of the member variables and that of τ, satisfy this specification. In standard approaches to reasoning about method behavior, no facilities for reasoning about method call traces are provided, so we introduce some appropriate techniques here.

Let us assume that we are working within an axiomatic style reasoning system with a full complement of standard rules and axioms. The difference between reasoning about the behavior of a template versus a non-template method is the need, in the latter case, to write assertions about the current value of the trace, τ. Since most statements do not affect the value of τ, the rules and axioms for them can be used with no change. The one statement that can affect τ is a call to a hook method $h()$, so we introduce the rule **R1** below for reasoning about this statement. In this rule σ represents the state of the object, i.e., the values of all member variables of the class, and abl (all but last) and last are mathematical functions on traces which when applied to τ return, all of τ save the last element, and the last element of τ, respectively:

R1. Hook Method Call Rule

$$\frac{\begin{array}{l} p \Rightarrow \mathsf{h}(\mathsf{x}).pre^{x}_{aa} \\ [(\exists\sigma', aa').[\ \ p^{\tau,\ \ \sigma,\ \ aa}_{\mathsf{abl}(\tau),\ \sigma',\ aa'} \wedge \mathsf{h}(\mathsf{x}).post^{\#\sigma,\ \#x,\ \ x}_{\sigma',\ \ aa',\ aa} \wedge \\ \qquad \mathsf{last}(\tau) = (h, \sigma', aa', \sigma, aa)\ ]\ ] \ \ \Rightarrow \ \ q \end{array}}{\{\ p\ \}\ \mathsf{this.h}(aa);\ \{\ q\ \}}$$

The rule has two antecedents which must be proven before the conclusion can be asserted. The first antecedent requires that p (which is an assertion summarizing what we know about the state immediately before h is invoked) must imply the pre-condition of h (with the actual parameter aa substituted for the formal parameter x). The second puts some conditions on the assertion q which describes what we know about the state immediately after h returns. In particular, we know that whatever p asserted about τ, is now only true of the part of τ that does not include the last element, since this element was newly added as a result of the call to (and return from) h. Further, this new element consists of the name of the called method (h), the state (σ') immediately before the call, the initial value (aa') of the parameter, the state (σ) immediately after the return from h, and the final value of the parameter (aa). The rest of the conditions on q just require that the state after the return from h is related to the state before the call by the post-condition of h. Note that this refers to the post-condition that is provided with the framework, not any application-specific post-condition (we do not have any applications yet, so of course we do not have any application specific post-conditions for $h()$ either).

How can we be sure that this reasoning remains valid when the application designer redefines the hook method $h()$? This is what behavioral subtyping ensures, and will be guaranteed by our next rule. We first introduce some notation: Let $\mathcal{F}$ denote the framework and $\mathcal{A}$ the application we are interested in. Let $\langle h.f.pre, h.f.post \rangle$ denote the pre- and post-conditions of the hook method h as specified in the framework, and $\langle h.a.pre, h.a.post \rangle$ the pre- and post-conditions we wish to establish for h in the application. We will use $\mathcal{F} :: \langle h.f.pre, h.f.post \rangle$ to denote that this specification has been established for the framework F, and a similar notation for the application.

R2. Behavioral Subtyping (for Hook Methods)

$$\frac{\begin{array}{l} \mathcal{F} :: \langle h.f.pre, h.f.post \rangle \\ \{h.a.pre\}\ \mathcal{A}.\mathcal{H}\ \{h.a.post\} \\ h.f.pre \Rightarrow h.a.pre,\ \ h.a.post \Rightarrow h.f.post \end{array}}{\mathcal{A} :: \langle h.a.pre, h.a.post \rangle}$$

The first antecedent requires the appropriate specification to have been established for the framework. $\mathcal{A}.\mathcal{H}$ is the body of the hook method as defined in the application; the second antecedent requires us to reason about this body using the application-specific pre- and post-conditions. The third antecedent imposes the behavioral subtyping requirements. In the interest of keeping the presentation simple, we have not allowed for parameters to h in this rule, but it would be straightforward to do so. Also in the interest of simplifying the presentation, we have not included class invariants. Class invariants are especially important if $h.a.pre$ imposed any conditions on member variables introduced in the derived class, since $h.f.pre$ cannot ensure such conditions; in the paper that will include full formal details, we will also include invariants.

Let us now turn to our final, and most important rule, the one that will allow us to establish an application-specific behavior for a template method t by plugging in, into the framework specification of t, the application-specific behaviors of the hook methods that t invokes. This "plugging-in" can often be done informally by an application designer as long as a few key facts are kept in mind: First, the behavior of the hook method as defined in the application must be consistent with its behavior specified in the framework; this is the behavioral subtyping condition imposed by **R2**. Second, the application-specific data members introduced in a derived class cannot be affected by t except by calls to hook methods; in other words, the only changes that happen to the values of these data members during the execution of t are because of calls that t makes to the hook methods defined in the derived classes. Third, the state immediately before a call to a hook method h and the state immediately after the call will be related by the application-specific post-condition for h. It is this last point that will allow us to establish an application-specific behavior for t since *h.a.post* will be stronger than *h.f.post*, and will include, in particular, information about the new data members introduced in the derived class.

The rule **R3** formalizes these notions. We use σ denote the state of the application object, i.e., the (current) values of all the member variables defined in the framework (that is the base class), as well as those defined in the application (the derived class). $\sigma{\downarrow}(F)$ will represent the framework portion of σ; and $\tau{\downarrow}(F)$ represents the trace obtained by projecting out the framework portion of each of the states recorded in τ.

R3. Rule for Plugging-in Application

$$\frac{\begin{array}{l} \mathcal{F} :: \langle t.f.pre, t.f.post\rangle \\ t.a.pre \Leftrightarrow t.f.pre \\ [\ t.f.pre^{\sigma,}_{\#\sigma\downarrow(F),} \wedge\ t.f.post^{\#\sigma,}_{\#\sigma\downarrow(F),}{}^{\sigma,}_{\sigma\downarrow(F),}{}^{\tau}_{\tau\downarrow(F)} \wedge \\ \quad TBR(\#\sigma,\sigma,\tau)\ ]\ \Rightarrow\ t.a.post \end{array}}{\mathcal{A} :: \langle t.a.pre, t.a.post\rangle}$$

The first antecedent requires us to have established, in the framework, the appropriate specification for t. The second requires the framework-level pre-condition of t to be equivalent to its application-specific pre-condition. The next antecedent is the crucial one; it will, as we will see next, justify *t.a.post*. Given these three antecedents, **R3** allows us to derive, as its conclusion, the specification for t that is appropriate to this application.

We now consider the third antecedent. Recall that our post-conditions involve both the state (denoted $\#\sigma$) at the start of the operation as well as the state when the operation finishes; plus, of course, this being a template method, its specification will also include references to its trace τ. The first conjunct on the left side of this implication simply tells us that in *t.a.post*, we may include information about the initial state, i.e., the state when t begins execution, provided to us by *t.f.pre*, except that where *t.f.pre* refers to the state, we must refer to that state using the '#' notation. Further, the state that *t.f.pre* refers to is not the complete state but just the framework portion; hence we replace σ in *t.f.pre* by $\#\sigma \downarrow (F)$. Similarly the second conjunct allows us to include, in *t.a.post*, information about the initial and final states provided by *t.f.post* (with substitutions of $\#\sigma \downarrow (F)$ for $\#\sigma$ and $\sigma \downarrow (F)$ for σ).

The third conjunct on the left side of the implication tells us that further information, provided to us by the *TBR* relation, may also be included in *t.a.post*; indeed this is the part that will allow us to add application-specific information to *t.a.post*, and we consider that now. The *Trace Behavior Relation (TBR)* simply captures the facts that the information recorded in the elements of τ which represent calls to and returns from the hook methods is consistent with the specifications of these methods as defined in the application, and that the only change in the state, as far as the portion that is introduced in the application is concerned, is because of these calls. (We will use the notation $\sigma \downarrow (A)$ to denote the state introduced in this derived class paralleling the $\sigma \downarrow (F)$ notation which we use to denote the framework portion of the state.)

$$\begin{array}{l} TBR(\#\sigma,\sigma,\tau) \equiv \\ \quad (\forall k \in \{0,\ldots,|\tau-1|\}). \\ \quad\ [(k=0) \Rightarrow (\tau_k.is{\downarrow}(A) = \#\sigma{\downarrow}(A)) \wedge \\ \quad\quad (k=|\tau|-1) \Rightarrow (\tau_k.fs{\downarrow}(A) = \sigma{\downarrow}(A)) \wedge \\ \quad\quad (0 < k \leq |\tau|-1) \Rightarrow \\ \quad\quad\quad (\tau_k.is{\downarrow}(A) = \tau_{k-1}.fs{\downarrow}(A)) \wedge \\ \quad\quad (\tau_k.n = h_i(x)) \Rightarrow \\ \quad\quad\quad (h_i(x).a.post^{\#\sigma,}_{\tau_k.is,}{}^{\#x,}_{\tau_k.ip,}{}^{\sigma,}_{\tau_k.fs,}{}^{x}_{\tau_k.fp})] \end{array}$$

$|\tau|$ denotes the length, i.e., the number of elements in τ. τ_i is the ith element of τ, the elements being numbered starting at 0. $\tau_i.n$ is the name of hook method being invoked in the call represented by this element. $\tau_i.is$ is the state immediately before this call, $\tau_i.ip$ the value of the parameter passed to the call, $\tau_i.fs$ the state immediately after the call returns, and $\tau_i.fp$ the final value of the parameter. Thus the first clause of *TBR* says that the portion of state that was introduced in the application must be the same at the time of the first call to a hook method as it was when t was called since the only way that this portion of the state can change is as a result of a hook method being executed. Similarly the second clause states that this portion of the state does not change after the last call to a hook method has completed. The third clause says that this portion of the state is not changed in between calls to hook methods, and the fourth clause says that the state recorded in τ immediately before a call to a hook method h_i and the state recorded in τ immediately after that call returns must be related by the application-specific post-condition for h_i. It is this final clause that really allows us to plug-in the application specific behavior of the hook method h_i into the behavior of t and enables us to include, in *t.a.post*, information about how the portion of the state introduced in the application changes as a result of the execution of $t()$. It may seem that there ought to be an analogous clause that requires that the state at the time of the call to h_i satisfies $h_i(x).a.pre$. But such a clause is not needed since the behavioral subtyping requirement imposed by **R2** ensures that $h_i(x).a.pre$ will be satisfied at the time of the call, given that $h_i(x).f.pre$ must have been checked as part of the reasoning to show that the body of t, defined in the framework, satisfies its (framework) specification.

5 Example

Recall the BankAccount class defined in Figure 1. The framework-level specification of the template method NewMonth may now be written as follows:

$$\begin{aligned}
&\textit{NewMonth().f.pre} \equiv \textit{true}\\
&\textit{NewMonth().f.post} \equiv\\
&\quad (\textit{bal} = \#\textit{bal} + \Sigma_{i=0}^{\textit{nmonthlies}-1}\ \textit{monthlies}[i]) \wedge\\
&\quad !\{\textit{nmonthlies, monthlies}\} \wedge\\
&\quad (|\tau| = \textit{nmonthlies}) \wedge\\
&\quad (\forall i, 0 \le i < \textit{nmonthlies}.\\
&\qquad ((\textit{monthlies}[i] > 0) \Rightarrow\\
&\qquad\quad (\tau_i.n = \textit{Deposit} \wedge \tau_i.p = \textit{monthlies}[i])) \wedge\\
&\qquad ((\textit{monthlies}[i] \le 0) \Rightarrow\\
&\qquad\quad (\tau_i.n = \textit{Withdraw} \wedge \tau_i.p = -\textit{monthlies}[i]))\\
&\quad) \wedge\\
&\quad (\tau_0.is(\textit{bal}) = \#\textit{bal}) \wedge (\tau_{|\tau|-1}.fs(\textit{bal}) = \textit{bal}) \wedge\\
&\quad (\forall i, 0 \le i < \textit{nmonthlies}-1.\ \tau_i.fs(\textit{bal}) = \tau_{i+1}.is(\textit{bal}))
\end{aligned}$$

The first two clauses of NewMonth.f.post summarize the effect of NewMonth on the base class variables based on the behaviors implemented by the Deposit and Withdraw methods defined, in the framework. The third clause says the number of elements in τ is equal to the number of transactions in the monthly transaction array. The fourth clause specifies the identity of the hook method at position i in τ ($\tau_i.n$): Deposit or Withdraw depending on the whether the corresponding element in the monthly transaction array is positive or not. This clause also specifies that the parameter value ($\tau_i.p$) passed to the called hook method is the same as the (absolute) value at position i in the monthly transaction array. The fifth clause asserts that the state immediately prior to the hook method call recorded in position zero of τ, has a value for bal which is the same as its value when NewMonth was invoked, in other words, NewMonth does not modify bal before its first call to a hook method. The sixth clause says that the value of bal in the state immediately after the last call to a hook method returns, is the same as its value when NewMonth ends; that is, NewMonth does not modify bal after the return from the last hook method call. The final clause essentially says that not only does NewMonth preserve bal before the first hook method is called and after the last hook method returns, but also between calls to hook methods. It does this by stating explicitly that the value of bal in the final state recorded in position i of τ is the same as its value in the initial state recorded in position $i + 1$.

The final four clauses above may seem complicated, but they are important in capturing the interaction between NewMonth and the hook methods and will allow the application designer to understand the effect that particular redefinitions of the hook methods will have on the behavior of NewMonth. Before considering such redefinitions however, let us specify their behavior as implemented in the framework:

$$\begin{aligned}
&\textit{Deposit(amt).f.post} \equiv (\textit{bal} = \#\textit{bal} + \textit{amt}) \wedge\\
&\quad !\{\textit{nmonthlies, monthlies, amt}\}\\
&\textit{Withdraw(amt).f.post} \equiv (\textit{bal} = \#\textit{bal} - \textit{amt}) \wedge\\
&\quad !\{\textit{nmonthlies, monthlies, amt}\}
\end{aligned}$$

The pre-condition for each method is just *true*.

Let us now consider an application such as the PromoAcct discussed in Section 3. Suppose that in order to encourage customers to maintain high balances, the amount of miles awarded on a particular transaction will be proportional to the balance in the account. It is straightforward to implement such behavior in the derived class PromoAcct but rather than presenting the code for doing so, we just specify this behavior:

$$\begin{aligned}
&\textit{Deposit(amt).PromoAcct.post} \equiv (\textit{bal} = \#\textit{bal} + \textit{amt}) \wedge\\
&\quad (\textit{miles} = \#\textit{miles} + ceil(0.01 * \textit{bal})) \wedge\\
&\quad !\{\textit{nmonthlies, monthlies, amt}\}\\
&\textit{Withdraw(amt).PromoAcct.post} \equiv (\textit{bal} = \#\textit{bal} - \textit{amt}) \wedge\\
&\quad (\textit{miles} = \#\textit{miles} + ceil(0.01 * \textit{bal})) \wedge\\
&\quad !\{\textit{nmonthlies, monthlies, amt}\}
\end{aligned}$$

Next, we must turn to the following tasks:

1. Make sure that the new ('PromoAcct') behaviors of Deposit and Withdraw are consistent with their framework-level specification, as required by **R2**.

2. "Plug-in" the PromoAcct behaviors into the framework-level specification of NewMonth to obtain, by using rule **R3**, the richer behavior of that method specific to PromoAcct.

The first task is immediate since the PromoAcct-level specifications for Deposit and Withdraw simply provide additional information (concerning how miles, introduced in PromoAcct is updated) beyond that provided by the framework-level specification.

The second task requires some analysis. Our goal is to arrive, using rule **R3**, at a simple description of the effect that NewMonth has on miles (we already know its effect on bal). According to **R3** (and the auxiliary relation *TBR*), since miles is defined in PromoAcct, the only way it can be affected by NewMonth is through calls to the hook methods. Therefore, let us consider the series of states immediately before and after each call to a hook method during the execution of NewMonth, and these are recorded in its trace τ. From the rule **R3**, we know that the value of miles in the state before the call corresponding to the first element of τ is the same as its value when NewMonth starts, and the value of miles in the state after the last call is the same as its value when NewMonth ends. More formally,

$$\tau_0.is(\mathsf{miles}) = \#\mathsf{miles} \wedge \tau_{|\tau|-1}.fs(\mathsf{miles}) = \mathsf{miles}$$

Furthermore, **R3** also tells us that each time a hook method is called miles is changed according to the PromoAct post-condition of the hook method. Examining these assertions we see that each such call increases miles by one percent of the balance (rounded up); i.e.,

$$\forall i, 0 \leq i \leq |\tau|. \; [\, \tau_i.fs(\mathsf{miles}) = \tau_i.is(\mathsf{miles}) + ceil(.01 * \tau_i.fs(\mathsf{bal}))\,]$$

Therefore, in order to figure out the relationship between the values of miles at the start and end of NewMonth, we need to know the value of bal in each of the states in τ. For this we can turn to the specifciation of NewMonth to see that each time Deposit is called its parameter is the value in the corresponding position of the monthly transaction array, and each time Withdraw is called its parameter is the negative of the value in the corresponding position in the array. In effect, for a call at position i of τ, monthlies[i] is added to bal; i.e.,

$$\forall i, 0 \leq i \leq |\tau|. \; \tau_i.fs(\mathsf{bal}) = \tau_i.is(\mathsf{bal}) + \mathsf{monthlies[i]}$$

We also know from the framework-level specification of NewMonth (the sixth clause) that bal does not change in between calls to hook methods. This gives us enough information to show (by a simple induction) that the value of miles after the call to the hook function in position i of τ is given by:

$$\tau_i.fs(\mathsf{miles}) = \#\mathsf{miles} + \Sigma_{j=0}^{i} ceil(0.01 * (\#\mathsf{bal} + \Sigma_{k=0}^{j}\mathsf{monthlies}[k]))$$

Replacing i with $|\tau| - 1$ gives us the final value of miles in NewMonth, and this expression, along with the clauses from the framework specification of NewMonth regarding the value of bal, leads to the following application-level specification of NewMonth:

$$\textit{NewMonth().PromoAcct.pre} \equiv \textit{true}$$

$$\begin{aligned}&\textit{NewMonth().PromoAcct.post} \equiv\\ &\quad (\textit{bal} = \#\textit{bal} + \Sigma_{i=0}^{\textit{nmonthlies}-1}\ \textit{monthlies[i]}) \wedge\\ &\quad !\{\textit{nmonthlies, monthlies}\} \wedge\\ &\quad \textit{miles} = \#\textit{miles} +\\ &\qquad \Sigma_{i=0}^{\textit{nmonthlies}-1}\mathsf{ceil}(0.01 * (\#\textit{bal} + \Sigma_{j=0}^{i}\textit{monthlies[j]}))\end{aligned}$$

We have omitted, from this specification, information about τ because we do not expect the PromoAcct application to be further specialized; in *Java*-terminology, Deposit and Withdraw are *final* methods in PromoAcct. If this is not true, we would need to include that information too. In fact, in this case it would be useful to introduce two distinct specifications, one that contains information about τ, for use by designers who wish to further extend PromoAcct, the other (which is the one we have specified above) for use by clients of PromoAcct.

It is worth noting that our reasoning was indeed incremental: In arriving at the PromoAcct specification of NewMonth, in particular in determining its effect on miles, we did not have to reexamine the code of NewMonth. Furthermore, we also did not have to reestablish the effect of NewMonth on the framework-level variable bal since that information was already given to us by the framework-level specification of NewMonth. It is true that we had to go through a non-trivial amount of effort in combining the information provided by NewMonth.f.post concerning the states at the times of the calls to the hook methods, with the information from application-level specifications of these methods, to arrive at NewMonth.PromoAcct.post; and it may be claimed that compared to this, the savings in effort resulting from not having to reexamine the code of NewMonth is negligible. While there is some truth to this claim, it is a direct result of the simplicity of this example. In other words, the body of NewMonth literally consists of nothing more than calls to the hook methods. In realistic template methods, we would expect there would be considerable additional code; in such cases, the savings resulting from not having to reexamine that code will be substantial. And in some situations –such as if we purchased the framework from a software vendor and do not have access to that code– it may not even be possible to reexamine that code. Hence our approach would be essential in such situations.

6 Discussion

In her seminal paper [6] on reasoning about OO systems, Liskov states "A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of another type (the supertype) plus *something extra* [emphasis added]." Most of the work on the formalization of reasoning about the behavior of OO programs [7, 8, 2, 9, 5] of the last ten or more years has used the approach of behavioral subtyping, focusing on ensuring that 'subtype ... objects provide the behavior of ... the supertype'. For applications based on OO frameworks, we have argued, it is not sufficient to ensure that the derived class behavior is consistent with the behavior in the base class. It is also necessary to be able to reason about the ... *something extra* ... that the template methods exhibit in their behavior as a result of the richer behavior provided by the hook methods defined in the derived classes. Without this, all applications built on a framework would be equivalent to each other, and the framework approach would lose its power. This became clear even in the simple BankAccount example we presented.

The example also suggested the way to enrich the specification of the template methods so that the richer behavior of the hook methods implemented in the application could be plugged into it to arrive at the richer behavior of the template method. The key was to introduce, for each template method, a 'hook method call trace' that records information about the calls that the template method makes to the hook methods. And include in the specification of the template method, assertions relating the elements that appear in this trace and the values of the member variables of the class (and the parameters and results, if any, of the methods). The application developer could then plug into this specification the behavior implemented by his definitions of the hook methods to arrive at the richer behavior of the template method that clients of his application would see. Our rules were direct formalizations of these two operations.

The specifications, in our approach, of template methods can become somewhat complex. We believe that this is a reflection of the fact that frameworks are rather rich structures and that is what allows application designers to build applications tailored to their particular needs. The ability to do this depends upon having full information about the sequence of calls to hook methods that the template method will make, and exploiting this information. Our trace-based specification simply provides the notation for representing this information formally. We believe, however, that some simplification will be possible by utilizing special notations –such as regular expressions– to represent frequently occurring patterns in traces, and we intend to explore this in future work.

Another problem that we intend to address in our future work is rather more difficult. This has to do with *aliasing*, with two or more variables referring to the same object. Some progress, see for example [5], has been made in the literature to dealing with this problem outside the context of frameworks; but it is also important in the setting of frameworks because practical frameworks and applications built on them tend to make extensive use of aliasing. It remains to be seen whether the approach we have developed in the current paper can be combined with these techniques for dealing with aliasing.

We conclude with a brief comment about the relation of our approach to specifying frameworks with that of Campbell and Islam [1]. They consider the problem documentation of frameworks, but they focus on what might be called *structural* questions, such as identifying which objects are related to which other objects in the system, specifying constraints on (instances of) which classes may be combined with (instances of) which other classes, etc., whereas our focus has been on behavioral questions. For complete documentation of a framework, we must however deal with both types of questions.

References

[1] R.H. Campbell and N. Islam. A technique for documenting the framework of an object-oriented system. *Computing Systems*, 6:363–389, 1993.

[2] K.K. Dhara and G.T. Leavens. Forcing behavioral subtyping through specification inheritance. In *ICSE-18*, pages 27–51. Springer-Verlag, 1996.

[3] M.E. Fayad and D.C. Schmidt. Special issue on OO application frameworks. *CACM*, vol 40, 1997.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable OO software*. Addison-Wesley, 1995.

[5] K. Lano. *Formal Object-Oriented Development*. Springer, 1995.

[6] B. Liskov. Data abstraction and hierarchy. In *OOPSLA '87, Addendum to the proceedings*, pages 17–34. ACM Press, 1987.

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

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

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

[10] B. Foote R. Johnson. Desigining reusable classes. *Journal of OOP*, 1:26–49, 1988.