Understanding Frameworks

Neelam Soundarajan

Computer and Information Science
The Ohio State University
Columbus, OH 43210, USA

e-mail: neelam@cis.ohio-state.edu
Tel: (614) 292 1444; FAX: (614) 292 2911

June 1, 1998

Abstract

When designing a framework $\mathcal{F}$, the designer abstracts away certain key methods of certain key classes of the framework leaving them as virtual functions (in *C++* terminology). One of the most important contributions that $\mathcal{F}$ makes to the final application built on $\mathcal{F}$ is the *flow of control* implemented in the non-virtual functions of $\mathcal{F}$; this relieves the application builder from having to worry about this important and difficult issue. An application builder who builds a complete application $\mathcal{A}$ using $\mathcal{F}$ refines the abstract base classes of $\mathcal{F}$ by providing specific bodies for the virtual functions, thus implementing specific behaviors. We use the term *behavioral refinement* to characterize this type of refinement. The important questions we address in this chapter are the following: how do we understand the behavior –specifically the flow-of-control behavior– implemented by the framework code, how do we understand the behavior implemented by the method bodies defined in the application code, and how do we combine these to obtain an understanding of the behavior of the entire application?

We propose a *trace based* approach for specifying the behavior of the framework, in particular the control flow. The particular refinement that an application builder implements is captured in an appropriate refined specification of the respective virtual functions of $\mathcal{F}$. We then show how this refined specification can be combined with the specification of the framework to arrive at the behavior of the entire application. We illustrate our approach on a simple diagram editor framework.

Keywords: Specification and Verification, Traces, Flow-of-control Behavior, Behavioral refinement.

1 Introduction and Motivation

Frameworks [10, 18] promise to dramatically reduce the time and effort needed to develop complete applications. When designing a framework $\mathcal{F}$, the designer identifies certain key methods of certain key classes as *virtual* or *pure virtual* functions (in *C++* [19] terminology[1]). Although these methods will be provided definitions by the application developer, $\mathcal{F}$'s designer usually has some ideas on what kinds of behaviors these methods should exhibit such as which methods should be invoked at what points. This *flow of control* is what the framework designer implements in the (non-virtual) methods of $\mathcal{F}$ and is one of the main contributions that $\mathcal{F}$ makes to the various applications that might be built using $\mathcal{F}$.

The main task of a developer who wishes to build an application $\mathcal{A}$ using the framework $\mathcal{F}$ is to provide definitions for the virtual methods of $\mathcal{F}$. Let $\mathcal{A}'$ be the code that the application builder provides. Thus $\mathcal{F}$ and $\mathcal{A}'$ together constitute the complete application; we will denote this by writing $\mathcal{A} = \mathcal{F} \oplus \mathcal{A}'$. The application builder must have a good understanding of what $\mathcal{F}$'s designer expects of these methods and at what points they might be invoked. Unless the behaviors of the methods in the derived classes as implemented in $\mathcal{A}'$ by the application builder are consistent with the expectations of the framework designer, the application is unlikely to function properly. $\mathcal{F}$'s designer typically tries to express these expectations by choosing appropriate names for the methods and classes in question as well as informally explaining them in the framework documentation. Such informal expressions often provide adequate guidance to the application builder, especially in the case of modest sized frameworks for 'standard' application domains. But as the sizes of frameworks grow, and as frameworks for a variety of areas are developed, it is necessary to use more reliable methods for understanding the flow-of-control behavior implemented by the framework, for expressing the framework designer's expectations of the code in $\mathcal{A}'$, and for combining the framework behavior with appropriate information about the methods implemented in the concrete derived classes, to obtain a useful specification of the entire application. Our goal in this chapter is to develop such methods.

There are four distinct although related problems we must address in order to develop such methods. First, we must develop a precise approach and notation that can be used to specify the framework $\mathcal{F}$, including its flow-of-control behavior. This will tell us at what points the virtual functions of $\mathcal{F}$ might be invoked. Second, we need a procedure for combining this specification of $\mathcal{F}$ with appropriate information about the code in $\mathcal{A}'$, to obtain the specification of the complete application $\mathcal{A}$; the ultimate user will be interested only in this final specification since as a user he[2] is mainly concerned with how the entire $\mathcal{A}$ will behave, not the fact that it was built on the framework $\mathcal{F}$, nor questions about which part of the behavior exhibited by $\mathcal{A}$ comes from $\mathcal{F}$, and which part from $\mathcal{A}'$, etc. Third, we need a procedure that the framework designer can use to check that $\mathcal{F}$ does indeed meet its specification. And, fourth, the application builder needs a similar procedure to check that the code he has provided, i.e. $\mathcal{A}'$, meets its specification. We address each of these problems in section 3, after introducing a somewhat simple model of frameworks in the next section. Our focus in this chapter, and in particular in section 3, will be on the first two problems: developing ways to specify the framework $\mathcal{F}$, and to obtain the specification of the complete application $\mathcal{A}$.

The key idea underlying our approach to specifying the behavior of the framework $\mathcal{F}$ is to use *traces*, i.e., sequences of function calls and returns, to capture the control flow implemented by $\mathcal{F}$. As we said before, it is this flow of control behavior that is one of the key contributions that $\mathcal{F}$ makes. When trying to specify $\mathcal{F}$, we cannot expect to do so in terms of the precise *functional effect* of this sequence of calls since the methods being called are virtual; indeed the whole point of the framework is that different applications built using $\mathcal{F}$ will typically provide quite different bodies for these methods, with very different corresponding (functional) effects. What we need to do instead is to provide an abstract specification of how control flows among the various virtual methods, and that, as we will see, is exactly what our trace based approach allows us to do. Each element of the trace of a framework will represent a call to and the corresponding return from a virtual function; the details of what information is recorded in these elements we will postpone to section 3.

The framework will, in general, also provide functional behavior, specifically the behavior that should be

[1] We will generally use *C++* terminology and notation; but our approach is not in any way *C++* specific. We will use the terms 'function' and 'method' interchangeably.

[2] Following standard usage, we will use 'he', 'his' etc. as abbreviations for 'he or she', 'his or her', etc.

common to all applications built using this framework. We will use pre- and post-conditions in the standard Hoare-Dijkstra style to specify this aspect of $\mathcal{F}$'s behavior. Note that even virtual methods of $\mathcal{F}$ are defined in the framework and the specification of $\mathcal{F}$ will include their behaviors. Many languages, including *C++*, have the notion of a *pure* virtual function that does not have any definition associated with it in the base class. Although we will not consider such functions in the current chapter, it would be easy to extend our approach to include such functions. The behaviors associated with such functions in the framework's specification should then be considered as *constraints* that application designers must satisfy when they provide actual definitions for them.

One further point should be noted: throughout this chapter, we will be mostly working with what are usually called '*concrete specifications*', see for example [17], in other words our specifications of functions are in terms of their effects on the actual data members of the classes we are dealing with. When trying to understand the behavior of the framework and the application in terms of how they are built, naturally we have to work with such specifications since the behavior they exhibit is realized in terms of these components. Once we have an understanding of the concrete behavior of the entire application, we can use standard ways [11, 17] to convert it into an *abstract* specification in terms of a conceptual model.

The problems of using OO have been remarked on by many people in the literature see, for example, [5]. One of the important issues is that of documentation; the issue becomes particularly critical for frameworks as has been noted by a number of authors [3, 9, 2]. Most of these authors focus on informal documentations rather than the precise specifications of the kind we are interested in. We believe that both informal documentations, perhaps using examples as suggested by some authors, as well as precise specifications are useful and should be considered as complementing each other. Helm et al [7] use their formal notion of *contracts* to describe many relations between objects; however, their goal seems more towards using contracts for mechanical execution purposes rather than for documentation. We will return to the relation between the approach we propose and that of other authors in the final section of the chapter but one point is worth noting here: Approaches such as those of [11, 4, 13] that are used for specifying the behavior of 'normal' OO programs are not well suited for dealing with frameworks. The problem is that these approaches tend to downplay the contribution of virtual functions to overall system behavior. More precisely, suppose f is a virtual method in a class B; the usual approaches require us to provide a *sufficiently general* characterization of f such that definitions of f in the various derived classes of B all satisfy this characterization. Further, and this is what makes these approaches unsuitable for use with frameworks, the *only knowledge* that clients have regarding f is whatever is provided by this general characterization; the differences between the definitions of f in different derived classes of B are abstracted away. For frameworks this would mean that *all* applications built on a given framework would be *equivalent to each other* since the only differences between them is in how they define the virtual functions of the framework! Clearly we need a different approach to the specification of framework behavior, one that will allow us to distinguish between these applications.

The rest of the chapter is organized as follows: In the next section we present a simple model of frameworks that we will use throughout the chapter. Although not every framework will fit this model, it is general enough to illustrate the main issues that must be dealt with in documenting the behavior of frameworks. In the third section we consider how an application developer refines the framework $\mathcal{F}$ by providing appropriate definitions for the various virtual functions of $\mathcal{F}$, and contrast this type of *behavioral* refinement with the standard notion of *procedural* refinement. The key point goes back to the reason why standard approaches are not suitable for dealing with frameworks: the client is in general very interested in the added behavior implemented by the application developer, so it is not enough to simply assert that the new definitions of the virtual functions meet their original specifications, rather we must arrive at a richer specification corresponding to this added behavior. Next we introduce our notation for specifying the behavior of the framework, using traces for capturing the flow-of-control behavior. We then consider the question of how an application developer can combine the behavior of the framework specified in this manner with the behaviors exhibited by his definitions of the virtual functions to arrive at the behavior of the framework. In section 4 we briefly consider how our approach may be applied to a (simplified version of a) diagram editor framework; this is a fairly typical framework and is based on a framework developed in Horstmann's [8] text book. Section 5 summarizes our approach, briefly relates it to other approaches, and reiterates the importance of both formal and informal documentations in understanding frameworks.

2 A Simple Model of Frameworks

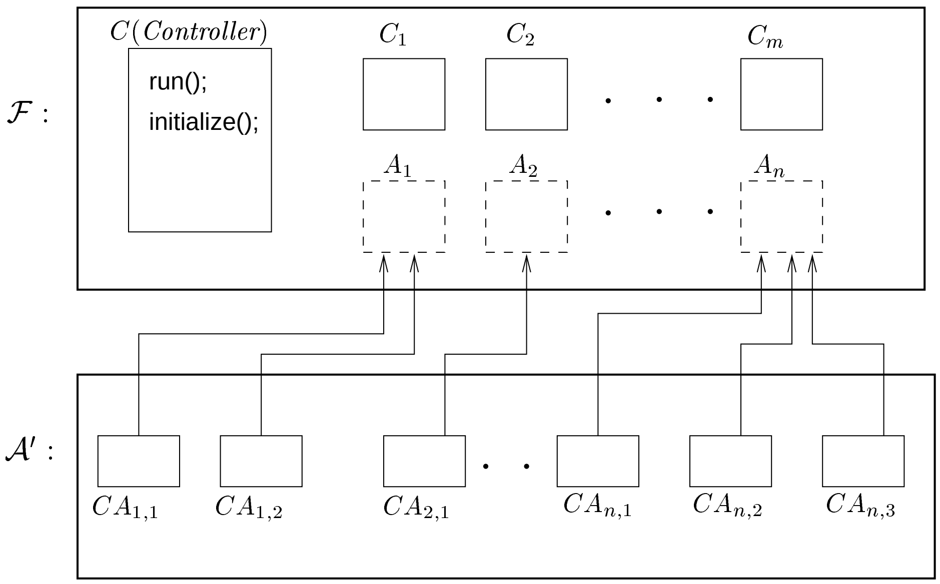
We will say that a class $\mathcal{A}$ is *abstract* if at least one of its methods is virtual. A class all of whose methods are non-virtual is a *concrete* class. A framework $\mathcal{F}$ will consist of the following classes:

- A concrete class C which we will also call the *Controller* class of $\mathcal{F}$. C will have a distinguished method named 'run'. As the name suggests, it is this method that primarily decides how control flows among the various methods of the various classes of $\mathcal{F}$.
- Zero or more other concrete classes $C_1, \ldots, C_m$.
- One or more abstract classes $A_1, \ldots, A_n$.

In addition to the run function, many frameworks also include a mechanism for initialization in order to allow some information about the actual application to be passed to the framework. Different frameworks use different ways to achieve this initialization.[3] In order to have a simple and uniform model, we will assume that the controller class C provides a single initialize function that will handle these tasks. To use the application built on $\mathcal{F}$ we, i.e., the 'main program', must first call the initialize function; when that finishes, we invoke the run function and start using the application.

An application $\mathcal{A}$ developed using $\mathcal{F}$ will have, corresponding to each abstract class A_j of $\mathcal{F}$, one or more derived classes $CA_{j,k}$ each of which will provide definitions for some or all of the virtual functions of A_j. We will use $\mathcal{A}'$ to denote this additional code provided by the application builder; and write $\mathcal{A} = \mathcal{F} \oplus \mathcal{A}'$. If definitions are provided in each of the derived classes in $\mathcal{A}'$ for all the virtual functions of the corresponding base classes of $\mathcal{F}$, the application will be a complete application; otherwise it will still be a framework, although a more concrete one than $\mathcal{F}$. Generally we will assume that definitions for all the virtual functions of $\mathcal{F}$ are being provided. For simplicity we will also assume that no new classes (that is classes that do not inherit from A_j) are introduced in $\mathcal{A}'$.

Pictorially the model may be represented as:



Dashed lines indicate abstract classes and solid lines indicate concrete classes. Arrows, as usual, indicate inheritance. The heavy solid lines enclosing the set of classes of $\mathcal{F}$ and $\mathcal{A}'$ do not correspond to any grouping

[3] Thus in the diagram framework of [8], each of the various classes of nodes and edges defined by the application code must be 'registered' with the framework before the run function is invoked; the diagram framework uses two functions, registerNode and registerEdge for this purpose.

mechanism of the programming language; perhaps future languages that explicitly cater to frameworks will have such mechanisms.

The 'main program' using the application will only have an object, let us call it d, of type C. d will be initialized first, followed by a call to run:

d.run();

Thus in the example of section 4, we consider a framework for dealing with a collection of figures. The controller class is CF; we have no other concrete classes; the single abstract class[4] is called F (for figure). In the application we consider, we will provide two concrete classes called C (for Circle) and T (Triangle) respectively as derived classes of F. In the main program, we declare a CF object d, initialize it and then run the application by calling run as shown above.

The objects of type C, such as d above, will have components of type $C_1, \ldots, C_m$ as well as $A_1, \ldots, A_n$ (or rather $CA_{1,1}, \ldots$); the main program itself will not have any objects of these types.[5] Inside the run function we will call the various functions defined in $C_1, \ldots, C_m$ and $A_1, \ldots, A_n$ (some of which will be virtual functions, and hence will call the corresponding functions defined in the appropriate concrete classes defined in $\mathcal{A}'$), to act upon these components of the main object d. In the example of section 4, the DC object will have two Figure objects of (possibly) different types. In the run function of DC, we will manipulate these components by invoking the various functions defined in the Figure class which will result, if the function we invoked was virtual, in the execution of the corresponding function body[6] defined in the appropriate concrete Figure class, Circle or Triangle respectively. Situations such as these are often called 'a template method (run) calling a 'hook' method (the virtual function)' [6].

Before concluding this section, we should note a couple of points regarding our model of frameworks. First, as we said before, and has been noted by several authors, perhaps the most important contribution that the framework makes is relieving the application developer from having to worry about how control should flow among the various methods of the various classes. In our model, this is highlighted by the run function. Second, many existing frameworks such as those for simulation of various phenomenon, to frameworks for GUIs, fit reasonably well within our model at least in a conceptual sense (although there may be syntactic mismatches; the control function may not be called run, for instance). Moreover, we believe that the main ideas underlying our approach to specifying and understanding frameworks, which are really the focus of this chapter, will also be applicable to frameworks that don't strictly follow our model, so no harm is done by using the simple model.

[4]The term 'abstract class' is commonly used to denote a class that has at least one *pure* virtual function. In this chapter though as we noted before, we do not take special note of pure virtual functions, focusing instead on virtual functions that have a definition in the base class, to be overridden by the definition that the application developer will provide. Thus we use the term abstract class to distinguish it from a concrete class which has no virtual functions, pure or otherwise.

[5]This is the reason for the earlier suggestion that perhaps languages should provide syntactic mechanisms that bundle the classes of $\mathcal{F}$ into a syntactic unit, identify the controller class C of $\mathcal{F}$, etc. That will make it clear that the 'client' program is only expected to have objects of type C, and that classes $C_1 \ldots, C_m$ etc. are of no direct relevance to the client.

[6]For simplicity we will follow the C++ convention that the number and types of the parameters of the virtual functions in the base class must match exactly the (number and) types of the parameters of the corresponding functions in the derived classes; other more general requirements such as *contra-variance* or *co-variance* are also possible [1], but we won't consider them in this chapter.

3 Behavioral Refinement: From Frameworks to Applications

The run function of $\mathcal{F}$ provides a 'behavioral skeleton' for the entire application with the virtual functions being '(partially) empty shells' in some sense. Suppose f is such a virtual function and it is a member of the abstract class A. The application builder will define the body of f when defining the concrete class or, as happens in the example in the next section, if there is more than one concrete class corresponding to A, then the body of f corresponding to each concrete class that inherits from A. Thus this designer *refines* the behavioral shell corresponding to f. But in doing so he must not violate the constraints imposed on these functions by the framework designer, in other words the redefined body of the virtual function must satisfy its specification as given in the specification of $\mathcal{F}$. In addition, and this of course is the point of using virtual functions, when defining the body of f the application developer will build in *additional functionality*, usually in terms of its effect on new member variables introduced in the concrete class.

A simple, if artificial, example will make the point clear. Suppose an abstract class in $\mathcal{F}$ has an integer member variable x. Suppose there is a virtual function changeX in this class. The framework designer may impose a constraint that says that corresponding to a call d.changeX(), the final value of the member x of d when changeX finishes execution must be greater than its value when changeX started execution, and the other members of d must have the same values as they did before the call. This can be expressed quite easily using standard pre- and post-conditions for changeX(). Thus using the notation #x to denote the value of x before the call under consideration, the post-condition of changeX() will be:

$$post \equiv (\mathsf{x} > \#\mathsf{x}) \tag{1}$$

What if the framework designer imposed a more severe constraint, something like

$$post \equiv (\mathsf{x} = \#\mathsf{x} + 1) \tag{2}$$

Doesn't this specify the effect of the function body precisely? Indeed, assuming that x is the only member variable of the class in question, it defines exactly the effect of changeX. 'Frameworks' that impose constraints such as (2), leaving as they do little or no freedom to the application developer, would hardly seem to deserve being called frameworks. But such is not the case; the point is that the added behavior that the application developer is usually interested in providing is in terms of the effect that the virtual function has on *new* member variables in the derived class in $\mathcal{A}'$ and the framework, of course, imposes no constraints on what changeX may or may not do to these new components. Thus the precise effects of the function bodies defined by the application builder, *including* the effects on any new member variables introduced by the application code $\mathcal{A}'$, will only be captured in the specification of $\mathcal{A}'$. This information will then be combined with the specification of $\mathcal{F}$ to obtain the behavior of the entire application, in particular of the run function.

We should note that while behavioral refinement bears some resemblance to the standard notion of *procedural refinement*, there is an important difference: As in procedural refinement, we must ensure that the definition that an application developer provides for a virtual method f meets its constraints, i.e., satisfies its specification in the framework. But unlike in procedural refinement, we are very interested in the *additional behavior* provided by this f and, what is more, this additional behavior leads to additional behavior being exhibited by the run function and we are interested in arriving at a corresponding richer specification for run. In other words, we are not just trying to verify that the particular refinement of f that the application developer has provided meets the constraints contained in the specification of the framework; we are also trying to arrive at a specification corresponding to the richer behavior of the entire application $\mathcal{A}$ given the particular refinement that the application builder has implemented.

3.1 Specifying the Framework

In this subsection we will consider how a framework $\mathcal{F}$ may be specified. In order to simplify the presentation, we will assume that we have only two classes in $\mathcal{F}$, a concrete class C and an abstract class A. C will contain only the run function which is non-virtual. C will contain a single member variable x of type A; it may also contain other member variables of pre-defined types (such as integers etc.). The class A will contain a non-virtual function f and a virtual function g. Note that f will not call g; allowing that would make it a template method calling a hook method like run and that would complicate the discussion because in specifying f we would have to use an approach similar to the one described below for specifying run. The function run will call g and may call f; the call to g will be of the form $x.g(\ldots)$. Allowing for more classes, more functions, and more member variables would not introduce any major conceptual complexities in the formalism, but we would have to introduce appropriate notational mechanisms for distinguishing between them etc. and these would add considerable complexity to the presentation.[7]

In order to specify $\mathcal{F}$, we must specify the behaviors of each of f, g, and run. The specification of f is perhaps the easiest. We just use standard pre- and post-conditions specifying respectively the conditions that the state, i.e, the values of the member variables, of A must satisfy when f is called, and the condition that the state will satisfy when f finishes execution. In our experience, the post-condition is best expressed as an assertion over the state at the time of the call to f and the state when f finishes rather than just the latter and we will use the '#' notation for this purpose. Thus if y is a member variable of A, then #y in the post-condition of f refers to value of y at the time of the call to f and y refers to its value when f finishes. Since f is non-virtual, it cannot be redefined by the application developer, thus the specification of f in the final application is the same as in $\mathcal{F}$.[8]

The specification of g is similar. But, of course, the application developer, when he defines a derived class corresponding to A will provide a new definition for g. We require that this new definition continue to satisfy the specification of g given as part of $\mathcal{F}$'s specification. This is required since in reasoning about run in $\mathcal{F}$, we will make use of this specification of g. In a sense, this specification of g expresses the constraints that the framework designer imposes upon the application developer. That developer may redefine g as he chooses so long as the behavior specified as part of $\mathcal{F}$ continues to be satisfied.

The most interesting function is, of course, run. Here again we use pre- and post-conditions but the post-condition will involve not just the state of the member variables (including x) of C when run finishes but also the trace of calls to g that run goes through during its execution. Given such a specification, the application developer will be able, as we will see in 3.2, to combine this with the specification of g as redefined in the application (that is in the derived classes of A) to arrive at the specification of run appropriate to his particular application.

Let us consider another simple but again artificial example. Suppose C contains only the single member variable x of type A. Suppose A has a single member variable v of type integer, and a single member function g which is virtual, receives no parameters, and that all it does is to increase the value of v by 1. The specification of g in $\mathcal{F}$ will be:

$$pre_{\mathsf{g}} \equiv true \tag{3}$$

$$post_{\mathsf{g}} \equiv (\mathsf{v} = \#\mathsf{v} + 1) \tag{4}$$

Suppose next that the body of run is as follows:

```
x.g(); x.g();
```

In the specification of run, we will use the symbol τ to denote the *trace* of calls it makes to the virtual function g; note that only calls to virtual functions are recorded in τ. Each element of τ corresponds to a single call to and the corresponding return from a virtual function. The element specifies the identity, i.e., the name of the function called, the object on which the function was applied, the state of the object at the time of the call and at the time of the return, as well as the values of any parameters to the function and any results returned by the function. In an actual specification we may include as much or as little of this information as we (as framework designers) choose. This is no different than writing specifications that omit information that is not of interest to the specifier about the function being specified.

[7] As we noted in section 2, one important function that the class C would include in most frameworks is an initialize function. One important task of this function is to assign the appropriate actual type, that is the identity of the particular derived class of A that x is an instance of. How this is achieved is somewhat language specific and we do not wish to go into these details here. In the example in the next section, we will just assume that by the time the main program invokes the run function, the actual types of all the member variables such as x have indeed been initialized.

[8] In *C++* as in many other OO languages, it is possible for a derived class to override non-virtual functions; we will ignore this possibility here since it seems to go against the philosophy behind frameworks. For one possible way of dealing with such overriding, but not in the context of frameworks, see [17].

The specification of run will be:

$$pre_{\mathsf{run}} \equiv \mathit{true} \tag{5}$$

$$post_{\mathsf{run}} \equiv (\mathsf{x.v} = \#\mathsf{x.v} + 2 \wedge (|\tau| = 2 \wedge \tau.o[1] = \tau.o[2] = \mathsf{x} \wedge \tau.f[1] = \tau.f[2] = \mathsf{g})) \tag{6}$$

The first clause in the post-condition of run asserts that when run finishes, the value of x.v will be 2 greater than at the start; the second clause asserts that during execution, run will call virtual functions twice ($|\tau|$ is the length of τ), and in each call the function that is called is g and the object on which the function is applied is x. Note that we cannot conclude, from this specification that the value of x.v when g is called the first time is the same as at the start of run nor that its value is one greater when g is called the second time. That is indeed the case, given the body of run that we wrote down above, but this information has not been included in this specification. For all we can tell from this specification, run might add 100 to x.v prior to each call to g and then subtract 100 from it after the return from g! Or alternately, perhaps run saves the initial value of x.v in some local variable and sets x.v to some completely arbitrary value before calling g twice in a row, and after the return from the second call resets x.v to its original value and then adds 2 to it.

But, we could, if we choose to do so, provide a stronger specification of run that would rule out these possibilities. All we would have to do is add to the specification the fact that the state of x at the time of the first call to g is the same as at the start of run and that the state of x at the time of the second call to g is the same as at the return from the first call. Would such added information be useful? Suppose the 'application developer' decides to add a new member variable, call it w, in the derived class D of A. Suppose also that the body he provides for g copies the value of v into w before incrementing v by 1 and then returning.[9] Then given the above specification of run, we would not be able to conclude that following a call to run the value of x.w is one greater than the value of x.v at the start of the call. With a stronger specification, we would indeed be able to arrive at this conclusion.

In general, if in the specification of the framework, especially in the specification of the trace of the run function, we leave out any information about which virtual functions are invoked, on what objects they are invoked, or what the states of these objects are at the time of these invocations, it is possible that an application developer will not be able to establish the complete behavior of the run function, specifically its effects on new member variables introduced in the derived classes he defines, for his particular application. The flip side is that including all of this information makes the specification relatively more complex. Thus the framework designer must, to an extent, anticipate what sorts of information are likely to be useful for the application developer and include all of that in his specification.

Two final points are worth noting. First, if run were a non-terminating function (as would probably be the case in most real frameworks), we would use invariants rather than post-conditions for specifying its behavior, but the basic ideas remain applicable. Second, while the idea of recording in a trace the calls made by the run function to virtual functions as well as the corresponding returns is fairly straightforward and even natural, the specifications that include this type of information can be quite complex even for simple run functions. In particular if run had a variety of possible sequences of actions that it will choose from and if these choices depend upon values returned by the calls to the virtual functions, it would be impractical to explicitly list all possible values of τ. We believe however that using appropriate formal notations, such as regular expressions, can considerably simplify these specifications; and we intend to pursue these possibilities in future work.

3.2 Specifying the Application

Let us now turn to the specification of the application. As we mentioned before, the application builder will define derived classes corresponding to the abstract classes of $\mathcal{F}$, and provide definitions, in these derived classes, for all the virtual functions of corresponding base classes. Continuing with the restricted model we used in 3.1, the application developer will provide one or more derived classes corresponding to A; in each of these classes the developer can introduce new member variables and must provide definitions for g. In order

[9] Note that incrementing v by 1 is *required* by the specification of g given earlier as part of the specification of the 'framework'; the code supplied by the application developer for g is required to satisfy this requirement.

to simplify the presentation in this section we will assume that only one derived class corresponding to A is introduced and we will name this class D. The definition of f will be inherited unchanged from A since f is not a virtual function.

The specification of f is also inherited unchanged. Or rather, it is strengthened slightly: we can be sure that the values of the new member variables introduced in D will be the same at the end of each call to f as at the start since these variables did not even exist when the body of f was written (as part of the base class A). For instance, if u were such a variable, the corresponding clause added to the post-condition of f would be $(\mathsf{u} = \#\mathsf{u})$.

The specification of g will also be stronger but for a different reason. The application developer has provided a new body for g and its new specification will correspond to this new body. But the new body of g must continue to meet the specification given for it by the framework designer. This is important since in reasoning about run, that designer has most likely relied upon this specification of g. If the new body of g did not satisfy this specification that reasoning would no longer be valid. Many OO languages, including $C{+}{+}$ allow the derived class definition of g to invoke the base class definition, so if the application developer wanted the effect of the new g on the variables of A to be the same as in the base class, he will only have to supply the code for manipulating the new member variables introduced in D. However g is implemented, the new specification for it will reflect the behavior of the new g, and will be a strengthening of its original specification.

The most interesting function is of course run. Because it calls g, and g has been redefined in the application to exhibit richer behavior, the behavior of run in the application will also be correspondingly richer. One possibility would be to just use the specification provided by the framework for run; this specification would still be valid (since g continues to satisfy the specification on which this specification of run is based). But this is not a satisfactory alternative since the power of frameworks derives from the richer behavior exhibited by run and in order for us to be able to reliably exploit this richer behavior it must be captured in a specification. A second possibility would be to arrive at a new specification corresponding to the richer behavior by *reanalyzing* the body of run, using information from the new specification of g when reasoning about the effect of calls in run to g during this reanalysis. This is also not a good alternative since it would be inconsistent with the basic philosophy of frameworks that you have to design, and by implication analyze, a framework only once, not once for each new application. In our approach we do not have to settle for either of these alternatives. We can instead use the trace based specification of run as provided by the framework designer, 'plug-in' the behavior of g corresponding to the calls to g recorded on the trace, and arrive at an enriched specification of run.

Consider again the example from 3.1. Suppose in the derived class D we introduce a new variable w and the redefined g increments this variable by 1 (in addition to incrementing v by 1 since it would not otherwise meet its original specification). Thus the new post-condition of g would be:

$$post_{\mathsf{g}} \equiv [(\mathsf{v} = \#\mathsf{v} + 1) \wedge (\mathsf{w} = \#\mathsf{w} + 1) \tag{7}$$

Given this, and given (6), we can arrive at the following post-condition for run:

$$post_{\mathsf{run}} \equiv ((\mathsf{x.v} = \#\mathsf{x.v} + 2) \wedge (\mathsf{x.w} = \#\mathsf{x.w} + 2)) \tag{8}$$

This is admittedly a simple and artificial example, but the idea should be clear. Since the original specification of run includes information about which virtual functions are called, and in what order we can, when we redefine these functions in the applications, combine the information contained in this specification with information about the behavior of the redefined functions to see what richer behavior run will exhibit, in particular what effect it will have on the new member variables. Note also that in this particular 'application', what the redefined g did to the new member variable w did not depend upon the value of v. So the fact that the original specification of run did not specify the relation between the value of x.v at the start of run and at the time of either call to g did not prevent us from arriving at (8). In practice, as framework designers, we should make sure that the specification in the framework of run includes all information that potential application developers might need.

So far we have focussed attention on *specifications*. The enriched specification of run was obtained by using the information contained in (6), the framework specification of run, and combining it with the stronger post-condition (7) of g into it. This is a *verification* step and has to be justified by a formal proof rule. Similarly when verifying that the body of run as defined in the framework meets its specification, we will need appropriate axioms and rules; specifically, these axioms must be set up so that the effect of a virtual function call on the trace –of appending an element recording the identity of the function being called, the object to which the function is being applied, the state of the object at the time of the call and at the time of the return, and the values of any parameters passed to and results received from the function– is accounted for appropriately. We will omit the formal details of these rules; in [15] we deal with a related problem and the rules presented there can be tailored to deal with the current situation. One point that is worth noting here is the similarity of this situation with that in reasoning about the behavior of distributed programs. Trace based approaches are commonly used [14, 16] for dealing with such programs, a trace being associated with each process of such a program to record its interactions, i.e., communications, with other processes of the program. The axioms for dealing with such communication commands in this setting are similar to the axioms we need for dealing with virtual function calls in the current setting. Similarly the rules in the distributed program setting that allows us to combine the specifications of individual processes of a program to arrive at a specification of the complete program are similar to the rules we need for arriving at a strengthened specification of run given its (trace based) specification in the framework, and given the stronger specification for the virtual functions in the application.

4 Case Study: A Simple Diagram Editor Framework

We will briefly see how our approach may be applied to a simple framework. The framework we consider is based on one presented by Horstmann [8]; Horstmann's framework is for dealing with a collection of figures made up of *nodes* and *edges*, for moving the figures around, drawing and erasing them etc. To keep the discussion simple, and to avoid getting into graphics issues, we will consider a highly stripped down version of that framework.

Our framework will consist of a concrete controller class CF and an (abstract) class F (for 'figure'). Let us first consider F. The details of what a figure actually is, will of course not be specified in the framework; that is what the application developer will do when he designs the derived classes of F, based on the needs of the particular application. In F we provide some basic features that all kinds of figures will have. Let us start with the (protected) member variables of F

```
int x, y; // coordinates of 'anchor point'
int ic; // is figure currently 'iconified'?
```

Every figure has an 'anchor' point; for a circle, this might for instance be the center. If the value of ic is 1, that indicates that the figure is currently iconified.

Next consider the functions of F, all of which are virtual:

```
virtual void iconify(){ ic := 1; }
virtual void deIconify(){ ic := 0; }
virtual void move( int dx, int dy ){ x := x+dx; y := y+dy; }
virtual void blowUp( int f ){ ; }
virtual void blowDown( int f ){ ; }
virtual int isIn( int u, int v ){ return 0; }
```

iconify and deIconify are intended to do what their names suggest. In the class F all these can do is to set the value of ic to the appropriate value. The application developer must provide appropriate redefinitions for them in the derived class(es) so that they behave as desired for the particular type of figure in question. That is why these functions are virtual.

move will move the figure by dx, dy. But, of course it is not enough to just move the coordinates of the anchor point. The figure (or its iconified version) must be 'redisplayed' at its new location; that is why this function is virtual so the application developer can provide appropriate definition(s) for it.

blowUp and blowDown[10] expand and contract the figure by the specified factor. In F, there is really nothing for these functions to do (since they should not change the coordinates of the anchor point or whether the figure is currently iconified or not) so they have empty bodies. isIn checks whether the point with the specified coordinates u, v is *inside* the figure. This function is also virtual because the way to carry out this check very much depends upon the geometry of the particular figure and so the function must be defined by the application developer.

Let us consider the specification of F. As we saw in section 3, we need to provide pre- and post-condition specifications for each of these functions. The pre-conditions for all of them are *true*. In the specifications of the post-conditions, we often need to assert for several of the member variables that they are not modified by the function in question; we will use the ! notation for this purpose; thus the clause !(x,y)[11] in the specification of iconify says that this function will not change the values of x and y.

$$
\begin{aligned}
post_{\mathsf{iconify}} &\equiv (!(\mathsf{x},\mathsf{y}) \wedge (\mathsf{ic} = 1))\\
post_{\mathsf{deIconify}} &\equiv (!(\mathsf{x},\mathsf{y}) \wedge \mathsf{ic} = 0)\\
post_{\mathsf{move(dx,dy)}} &\equiv (!(\mathsf{ic}) \wedge (\mathsf{x} = \#\mathsf{x} + \mathsf{dx}) \wedge (\mathsf{y} = \#\mathsf{y} + \mathsf{dy}))\\
post_{\mathsf{blowUp(f)}} &\equiv (!(\mathsf{x},\mathsf{y},\mathsf{ic}))\\
post_{\mathsf{blowDown(f)}} &\equiv (!(\mathsf{x},\mathsf{y},\mathsf{ic}))\\
post_{\mathsf{isIn(u,v)}} &\equiv (!(\mathsf{x},\mathsf{y},\mathsf{ic}) \wedge (result \in \{0,1\}))
\end{aligned} \tag{9}
$$

All of these specifications are straightforward. For instance, the specifications of iconify and deIconify say that they do not change the values of x, y, and have the appropriate effect on ic. When the application developer redefines them in the derived class(es), he must ensure that the redefinitions continue to satisfy the specifications above. So, for instance, deIconify may not change the values of x, y and must set the value of ic to 0.

Similarly the application developer is at liberty to redefine move as he chooses so long as it makes the changes specified above to the values of x, y, and leaves the value of ic unchanged. blowUp and blowDown may be defined as we choose but they cannot change the values of any of x, y, or ic.

The specification of isIn is worth remarking upon. Note first the special symbol *result*; this is used (borrowing a convention from [13]) to denote the value returned by this function. This specification says that the value returned by isIn is either 0 or 1. Given that the body of isIn as defined in the code above actually returns 0, why this weak specification? Because if we strengthened it to, say, ($result = 0$), the application developer, when he redefines isIn will be forced to return this same value! In fact, in the class F, we have no way to decide whether or not the given point (u,v) is within the figure in question. So the value 0 being returned here was arbitrary; it is meant to be overridden by the correct value in the derived class. That is what the specification of isIn reflects.[12] In one respect this specification of isIn is weak; there is nothing to prevent the derived class designer from defining a body for isIn that returns 0 when the specified point is actually *in* the given figure and 1 when it is not; or even return 0 and 1 at random. But there is no way to include in the formal specification of F anything that represents our intuition that this function should check whether the given point lies within the figure or outside the figure and return 1 or 0 accordingly. That is why formal specifications such as ours should not be considered as replacing informal documentations but rather as complementing them.

Next consider the controller class CF. CF has two member variables f1, f2 both of type F. These are the figures that this 'framework' will let us work with. In a more realistic framework, such as the one in [8], the user of the application would be able to create as many figures as he wanted, not be stuck with two. But this would require us to handle (language dependent issues such as) creation of new objects etc., and we prefer

[10] shrink might have been a more conventional name for this function but blowDown was too tempting in its contrast with blowUp!

[11] The clause !(x,y) may be read as 'Don't touch x, y !'.

[12] It might perhaps be better to define isIn as a *pure* virtual function and not provide it any body in F. The application developer will then be forced to provide definitions for it in each derived class of F. Even if we do this though, it is important for the framework designer to provide a specification for isIn, in particular the part !(x,y,ic) since otherwise the application developer may not realize that isIn is not supposed to mess up the coordinates of the anchor point or whether a figure is iconified or not.

to avoid those questions by assuming a fixed number of figures. The run function will input a sequence of *edit-requests* and carry out each one. An edit-request will contain the following information: the coordinates of a point m, the identity of the edit operation to be performed –one of iconify, deIconify, move, blowUp, or blowDown–, and in case the operation is either blowUp or blowDown, the factor by which the figure should be expanded or contracted, or if the operation is move, the amount that the figure should be moved by. The point m may be thought of as the current *mouse* location. In other words, an edit-request gives us the current mouse location, as well as the desired operation, and in response run will carry out that operation on the figure at that location; if there is no figure at the given location, no action will be taken.

The code in run for processing an edit-request is fairly simple. Let us assume that the components of the request are: mx, my, the coordinates of the mouse position; op a character string (one of "Iconify", "DeIconify", etc.) that identifies the operation; and fac the factor if the operation is BlowUp or BlowDown, and dx, dy the distance to move the figure by if the operation is Move. All we need to do is check if the given mouse position is within either of the figures f1, f2 and if it is, invoke the appropriate operation on that particular figure. If it is within *both* figures, we apply the operation on f1; if it is within neither, we do nothing.

```
if ( f1.isIn( mx, my ) == 1 ) {
    if ( op == "Iconify" ) { f1.iconify( ); }
    else if ( op == "DeIconify" ) { f1.deIconify( ); }
    else if ( op == "..." ) { f1....( ); }
    ...}
else if ( f2.isIn( mx, my ) == 1 ) {
    if ( op == "Iconify" ) { f2.iconify( ); }
    else if ( op == "DeIconify" ) { f2.deIconify( ); }
    else if ( op == "..." ) { f2....( ); }
    ...}
```

The complete run function is just repeated execution of this code for each edit-request. Note the importance of the virtual functions in this code. In the framework, we do not have a complete implementation of isIn since that function depends on the type of figure we are dealing with. Nevertheless, and indeed this is the power of the framework approach, we are able to design the framework code to make use of this function, and call other functions based on the results returned by this function.

How do we specify run? We will just consider a single edit-request; dealing with a *sequence* of requests would simply mean repeating what we do when dealing with a single request. Indeed, this seems to be a common feature of most interactive applications, including those built on frameworks; this was the reason for our earlier suggestion that it may be useful to develop specially tailored notations that borrow ideas from regular expressions and other similar formalisms to simplify the specification of such systems. The particular request we consider corresponds to the Iconify operation; others are handled in a similar manner:

$$
\begin{aligned}
&{}^{post}\mathsf{run}(\mathsf{mx},\ \mathsf{my},\ \text{“}\mathsf{Iconify}\text{”}) \equiv \\
&\quad [\ !(\mathsf{f1.x},\ \mathsf{f1.y},\ \mathsf{f2.x},\ \mathsf{f2.y}) \\
&\quad \wedge ((|\tau| = 2 \vee |\tau| = 3) \wedge (\tau.o[1] = \mathsf{f1}) \wedge (\tau.f[1] = \mathsf{isIn(mx,my)}) \\
&\qquad \wedge ((\tau.res[1] = 1) \Rightarrow ((|\tau| = 2) \wedge (\tau.o[2] = \mathsf{f1}) \wedge (\tau.f[2] = \mathsf{iconify}) \wedge (\mathsf{f1.ic} = 1) \wedge (!(\mathsf{f2.ic})))) \\
&\qquad \wedge ((\tau.res[1] = 0) \Rightarrow ((\tau.o[2] = \mathsf{f2}) \wedge (\tau.f[2] = \mathsf{isIn(mx,my)}) \\
&\qquad\qquad \wedge ((\tau.res[2] = 1) \Rightarrow ((|\tau| = 3) \wedge (\tau.o[3] = \mathsf{f2}) \wedge (\tau.f[3] = \mathsf{iconify}) \\
&\qquad\qquad\qquad \wedge (\mathsf{f2.ic} = 1) \wedge (!(\mathsf{f1.ic})))) \\
&\qquad\qquad \wedge ((\tau.res[2] = 0) \Rightarrow ((|\tau| = 2) \wedge (!(\mathsf{f1.ic},\ \mathsf{f2.ic}))))))) \ ]
\end{aligned} \tag{10}
$$

This asserts that the coordinates of the anchor points of f1, f2 are not affected by carrying out this request; that run, in carrying out this request, makes either two or three virtual function calls; that the first call is to isIn, with the object being f1, and the argument being the point (with the coordinates) (mx,my) which was part of the original request; if the result of this call ($\tau.res[1]$) is 1, there is one more call to a virtual function, this one being iconify, the object being again f1. If the result[13] of the first call to isIn is 0, there is another call to isIn, the object this time being f2; here again the argument is the point (mx,my); if the result of this call is 0, there are no more virtual function calls, and the ic components of both f1 and f2 remain unchanged; if the result is 1, there is one final call, this time to iconify, the object being f2.

Consider now an 'application' built using this framework. We will have two concrete derived classes of F, the class C for Circles, and the class R for Rectangles. C will have two extra member variables:

```
int rad; // radius of circle
int count; // number of times iconified: to be explained shortly
```

R will have three extra member variables:

```
int length, width; // length and width of rectangle.
int color; // has value 1, 2, or 3 : to be explained shortly
```

The isIn function for C is easy to define; we compare the distance of the given point from the center, i.e., the anchor point, of the circle and if it is less than or equal to rad, the point is in the circle. But this is true only if the cicle is not currently iconified. We will assume that an iconified circle will be displayed as a small circle of radius *.5 cm*. Hence:

```
virtual int isIn( int u, int v ) {
    if ( (ic==0) && (dist(x,y,u,v)≤rad) ) return 1;
    if ( (ic==1) && (dist(x,y,u,v)≤.5) ) return 1;
    return 0; }
```

The function dist(x,y,u,v) returns the distance between the point (x,y) and the point (u,v). The code for isIn for the R class is similar and we will omit it.

Now consider the functions iconify and deIconify. In practice, both of these will change the state of the *screen*; since we do not want to get involved with graphics details regarding the screen, we will associate a different (and somewhat arbitrary) behavior with these functions in each of C and R.[14] Suppose it is difficult to draw circles, so the designer of the circle class wishes to keep a count of the number of times a circle is being iconified and deiconified. The following definition of iconify serves this purpose, assuming that count is initialized by the constructor function to 0; note also that if the circle is already iconified, we do not increment count:

```
virtual void iconify( ) { if ( ic==1 ) return; ic := 1; count++; }
```

Similarly, suppose the designer of R wishes to change the color of the rectangle each time it is iconified/deiconified, and that there are three colors to choose from, represented by the value of the color variable. Here is the code for iconify in the circle class:

```
virtual void iconify( ) { if ( ic==1 ) return; ic := 1; color := (color + 1) mod 3; }
```

These functions are easily specified. Thus the specification for iconify for the circle class is:

$$post_{\mathsf{iconify}} \equiv (!(\mathsf{x,y}) \wedge (\#\mathsf{ic} = 0 \Rightarrow \mathsf{count} = \#\mathsf{count} + 1) \wedge (\#\mathsf{ic} = 1 \Rightarrow \mathsf{count} = \#\mathsf{count}) \wedge (\mathsf{ic} = 1))$$

The specification of isIn for the circle class is a bit longer but not much more complex:

$$\begin{aligned} post_{\mathsf{isIn(u,v)}} \equiv [\ (!(\mathsf{x,y,ic,count}) \wedge (&(\mathsf{ic} = 0 \wedge distance((\mathsf{x,y}),(\mathsf{u,v})) \leq \mathsf{rad} \wedge result = 1) \vee \\ &(\mathsf{ic} = 0 \wedge distance((\mathsf{x,y}),(\mathsf{u,v})) > \mathsf{rad} \wedge result = 0) \vee \\ &(\mathsf{ic} = 1 \wedge distance((\mathsf{x,y}),(\mathsf{u,v})) \leq .5\ cm \wedge result = 1) \vee \\ &(\mathsf{ic} = 1 \wedge distance((\mathsf{x,y}),(\mathsf{u,v})) > .5\ cm \wedge result = 0)))\ ] \end{aligned}$$

where *distance* is the standard function for computing the distance between two points. We will the leave the similar specifications for the rectangle class to the reader.

The next step is to 'plug-in' these stronger specifications into (10) to obtain a stronger specification of run appropriate to this particular application. But first we need to know, for each of f1 and f2, whether it

---

[13] From the specification of isIn in the framework, we know that the only possible results of this call are 0 and 1.

[14] One thing we cannot change is what these functions do the variable ic since that is dictated by the specification of F. Nor can we define these functions to modify x, y since that is also forbidden by the specification of F.

is an instance of C or R. Once we have this information, we will be able to choose between the specification of, say, isIn from the class C or the class R when this function is applied to the object f1 or f2. In an actual framework, the initialize function of CF will allow the user to decide the class that each of these objects is an instance of. Here we will just assume that f1 is an instance of C and that f2 is an instance of R. This will allow us to strengthen, for instance, the clause

$$[(\tau.res[1] = 1) \Rightarrow ((|\tau| = 2) \wedge (\tau.o[2] = \mathsf{f1}) \wedge (\tau.f[2] = \mathsf{iconify}) \wedge (\mathsf{f1.ic} = 1) \wedge (!(\mathsf{f2.ic}))))$$

from the specification (10) of run as follows:

$$[(\#\mathsf{f1.ic} = 0) \wedge \mathit{distance}((\mathsf{f1.x,f1.y}), (\mathsf{mx,my})) \leq \mathsf{f1.rad}] \Rightarrow [(\mathsf{f1.ic} = 1 \wedge \mathsf{f1.count} = \#\mathsf{f1.count} + 1) \wedge \ldots]$$

This assures us that if the circle f1 is currently not iconified, and if the mouse position is within the distance rad of the center of f1, then following this call to iconify, f1 will indeed be iconified (and the count will be appropriately incremented). This is precisely the richer behavior that run acquires as a result of the definitions provided by the application developer in the class C, and we arrived at this without reanalyzing the code of the framework which was indeed our goal. Perhaps the most complex part of this task is working with the trace based specification of the run function. As we noted before, we hope the task will become considerably simpler if we can develop appropriate special notations for dealing with such specifications, and we intend to explore ways of doing this in future work.

5 Discussion

Object-Oriented frameworks can allow an application developer to develop, with relatively little effort, an entire application tailored to his particular needs. But this requires the application developer to have a good understanding of the framework, in particular he needs to know the behavior that the functions he defines in his derived classes are required to exhibit. There are various means by which the application developer can acquire this knowledge. Thus, for instance, if the framework designer chooses the names of the various methods carefully, that can convey a lot of information to the application developer. Studying existing applications developed using the framework in question is another important approach that the developer can use to develop an understanding of how the framework is meant to be used. We have proposed complementing these approaches by formally specifying the behavior of the framework.

Like most formal approaches, our specifications can be somewhat difficult to understand, especially in the beginning. But, again as with most formal approaches, the payoff is that one gets a precise understanding of exactly what requirements the functions defined by the application developer must meet. Consider again the example of the diagram editor framework of the last section. The name isIn conveys quite a bit of information about the particular function and one might question the need for a formal specification. But suppose an application developer decided to define a new type of figure corresponding to *donuts*. Should the isIn function return 0 or 1 if the given point is within the hole in the donut? This question cannot be answered by looking at the examples of figures like circle and rectangle because such a situation doesn't arise for these figures. Nor does the name of the function allow us to answer the question. But the specification of the function that we wrote down as part of the specification of the framework makes it clear that this decision is entirely upto the application developer since the only conditions that the framwork specification imposes on the behavior of this function are that it not change the values of any of x, y, ic, and that it return either 0 or 1 as the result. But at the same time this specification does not convey the intuition –as does its name– that this function is intended to tell us whether or not the given point is inside the figure in question. Thus the formal specification clearly complements other approaches such as suitable choice of names, or illustrative examples etc., rather than replacing these other approaches. The formal specification also makes it clear that should the application developer choose to ignore the intuition conveyed by the name of the function, as well as the guidance of the example applications, he may do so, as long as the requirements imposed by the formal specification are satisfied. This is important because occasionally the developer may find ways to use the framework in ways unintended by the framework designer or at least in ways different from those suggested by such things as the names of the functions as well as other applications that may have been developed using the framework.

While the approach we have proposed, in particular the use of *traces* to record the sequence of calls that the run function of the framework makes to the virtual functions of various abstract classes, is fairly natural, the resulting specifications are as we noted before rather difficult to read and understand. This seems primarily due to the lack of suitable notations for expressing fairly simple properties involving traces, so that we had to resort to low-level primitives (such as considering each element of the trace individually). We intend to develop appropriate notations that will make the expression of common behaviors easier to write and to read. This, we believe, will go a long way toward improving the usefulness of our specifications. It is also possible that we may be able to formalize ideas such as *scenarios* that have been used in informal approaches such as *OMT* to describe some of the flow-of-control information and use them in our approach.

Another extension that we need is to generalize our model of frameworks. The model we have considered in this chapter, while it seems to be a reasonable fit for many relatively simple frameworks, is not general enough. We intend to apply our approach to specifying an actual framework (for collecting and disseminating medical information of a particular type) developed by Mamrak *et al* [12]. We believe this exercise will allow us to generalize the model so that it will be suitable for realistic framework.

Finally, there is also the question of whether the programming language is too constraining. Thus, for instance, *C++* requires that the number and types of parameters received by a virtual function like isIn in the derived class be the same as that in the base class. This certainly makes the language, as well our model and the resulting specification issues, simpler but does it prevent us from building interesting frameworks? Isn't it conceivable that an application developer might want to develop a particular derived class for which the appropriate isIn function requires some additional parameters? How do we allow this (without, of course, abandoning type safety, etc.)? This is clearly a much more difficult question and we hope that attempting to generalize our specification technique to deal with such frameworks will shed some light on what needs to be done.

References

[1] M. Abadi and L. Cardelli. On subtyping and matching. In *Proceedings of ECOOP*, LNCS 952, pages 145–167. Springer-Verlag, 1995.

[2] J. Bosch and M. Mattsson. Framework composition. In Ege, Singh, and Meyer, editors, *Proceedings of TOOLS 23*, pages 203–214. IEEE Computer Society Press, 1997.

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

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

[5] M.E. Fayad and D.C. Schmidt. Special issue on object oriented application frameworks. *CACM*, 40, October 1997.

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

[7] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *OOPSLA-ECOOP*, pages 169–180, 1990.

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

[9] R. Johnson. Frameworks = components + patterns. *Comm. of the ACM*, 40(10):39–42, 1997.

[10] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object Oriented Programming*, 1:22–35, 1988.

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

[12] S. Mamrak, J. Boyd, and I. Ordonez. Building an information system for collaborative researchers. *Software Practice and Experience*, 27(3):253–263, 1997.

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

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

[15] S. Fridella N. Soundarajan. Enriching subclass specifications, technical report, available at *www.cis.ohio-state.edu/∼neelam*, 1998.

[16] N. Soundarajan. Axiomatic semantics of CSP. *ACM TOPLAS*, 6:647–662, 1984.

[17] N. Soundarajan and S. Fridella. Inheriting and modifying behavior. In Meyer Ege, Singh, editor, *Proceedings of TOOLS '97*, pages 148–163. IEEE Computer Society Press, 1997.

[18] S. Sparks, K. Benner, and C. Faris. Managing OO framework reuse. *Computer*, 29(9):52–62, 1996.

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