Use of Object-Oriented Programming in a Time Series Analysis System

R. K. Kerr* and D. B. Percival†

*Applied Physics Laboratory,
University of Washington, Seattle, WA 98105

†Applied Physics Laboratory and Department of Statistics,
University of Washington, Seattle, WA 98105

ABSTRACT

We describe the use of object-oriented programming (OOP) in the design and implementation of TSA, a system for interactive time series and spectral analysis. We show how such features of OOP as inheritance, generic messages, and decomposition of the programming problem in terms of objects have contributed to our goal of providing an extensible data analysis system. We discuss the strengths and limitations of both OOP and the particular implementation we used (Flavors on a Symbolics Lisp Machine) for our particular problem.

1. Introduction

We describe here the role that object-oriented programming (OOP) has played in the design and implementation of a system for interactive time series analysis. This system, called TSA, is currently implemented on a Symbolics Lisp Machine and is written in the OOP language Flavors. Our report summarizes the rationale for creating TSA (Section 2); the features of OOP that led us to use it for implementing TSA (Section 3); a description of the overall design of the system and how it appears to a user (Section 4); the use of OOP in an automatic programming facility that is useful for extending the capabilities of TSA (Section 5); and an evaluation of our experience to date with OOP (Section 6).

2. Rationale for Creating TSA

Time series analysis is primarily concerned with the characterization of dependencies in data (called a time series) that are gathered over a period of time. It is widely used in disciplines as diverse as economics (to analyze, say, weekly values of a forward exchange rate) and electrical engineering (to characterize the relationship between the input and output to a speech enhancement device).

Time series analysis is a rather complex field since it embodies ideas from Fourier analysis, signal processing, and statistics. Because of this inherent complexity, it is also a field where *interactive* data analysis is the rule rather than the exception. Interactive data analysis involves human intervention and judgement. It should be contrasted with *production* data analysis in which data is fed into a well-tested computer program and final results pop out. Interactive time series analysis typically involves iterations of cycles in which the analyst performs some operation on his or her time series (say, calculation of an estimate of its spectrum); carefully considers the results of this operation; and performs some new operation based upon the results of this examination (computation of a new estimate with different statistical properties). A key point to note here is that the next operation might well be implementation of an entirely new form of

analysis that is inspired by the time series at hand.

Our primary rationale for creating TSA was to develop a computer system that would support interactive time series analysis as fully as possible. There are a number of recent developments in computer hardware and software that, if properly harnessed, can lead to a substantial improvement in the quality of interactive time series analysis that an analyst can do. These developments include the emergence of affordable single-user graphics workstations with high resolution and a fast refresh rate; advances in the use of multiple windows and pointing devices (such as a mouse) to facilitate user interaction with a computer; and integrated programming environments.

How each of these developments can lead to better quality interactive data analysis is spelled out in series of recent articles by McDonald and Pedersen (1985a, 1985b, 1986). These authors note the close resemblance between experimental programming and interactive data analysis: both require the ability to make quick changes in software in response to creative ideas for solving particular problems. They argue that the integrated programming environments on specialized workstations such as Lisp Machines that were developed to support experimental programming are potentially good environments for doing interactive data analysis. There are certain things that are necessarily currently lacking in these environments, and one of the motivations for TSA was to supply the missing pieces for time series analysis.

A secondary rationale for creating TSA was to develop a system that is useful for helping students learn time series analysis. The ability to do good time series analysis is largely a matter of practice. The complexity of the field is such that students only really master the subject matter by applying textbook methods to many different time series. It is only by experimentation that they learn the proper use of standard techniques. The difficulty of teaching good data analysis is increasing as the complexity of the fields in which data analysis is important increases and as the number of tools available to an analyst increases. Indeed, John Tukey (1984), one of the founders of modern data analysis, has stated that the key education problem today in data analysis is developing courses that "· · · expose the innocent rapidly enough, without endangering whole subject-matter fields by producing not half-educated users (of which there are too many today), but rather only quarter-educated ones." The only bright hope that he sees for the future is the emergence of sophisticated computer software dedicated to particular aspects of data analysis (such as time series analysis). Such software would aid instructors in the teaching of data analysis, offer guidance and help to students in learning course material, and be powerful and portable enough so that students can carry it with them and use it for actual data analysis later in their careers. Hopefully TSA is a modest step in this direction.

Our design goals for TSA can be summarized as follows.

[1] The system should be accessible to the novice, but it should not hinder the experienced user in his or her tasks. This requirement rules out a simple nested menu scheme, since it would quickly drive an expert to distraction.

[2] It should support the commonly used techniques of time series analysis, but it should also be easily extensible to handle user-defined analysis procedures. It is the lack of extensibility that is the chief disadvantage of such batch-processing oriented statistical systems as SPSS and BMDP.

[3] It should be built upon a modern integrated programming environment and should itself be well integrated into that environment. In particular, it should make use of all of the tools of such an environment such as inspectors, the ability to evaluate expressions from within a text editor, etc.

[4] The overall structure of the program should be as transparent as possible to users. This is important if users are to be able to make the quick modifications demanded by interactive data analysis.

3. Why OOP?

When we started work on the TSA project in early 1985, we made a quick prototype of the system using procedure-oriented programming with Lisp functions. It soon became apparent that OOP was a more appropriate language paradigm to use for several reasons. First, it made sense to use OOP since a major portion of the software for the Lisp Machine was written in the OOP package Flavors. One of our design goals was to integrate TSA into the Lisp Machine programming environment so that all of the tools in that environment could be used for interactive data analysis when needed. As an example, an early goal of our project was to allow users to interact with graphs of their data using the mouse connected to the Lisp Machine. One desirable interaction is to be able to identify points on a graph by depressing a mouse button when the mouse cursor is close to the desired point. The way in which users make use of the mouse on the Lisp Machine is by receiving messages sent from a process that controls the mouse. Handling the mouse thus necessitates the use of OOP, as do all interactions with the Lisp Machine window system. Since TSA makes considerable use of the mouse and the window system, it was desirable to use OOP for the sake of uniformity.

A second initial reason for choosing OOP was our perception that the inheritance capabilities of Flavors might prove useful. Our early ideas were mainly focused on the creation of different types of graphs, some of which would be rather simple variations on more basic ones. In fact, inheritance has turned out to play a key role in the design of extensibility features for TSA (see Section 5).

Several other reasons for using OOP have emerged *a posteriori* in the process of creating and extending the capabilities of TSA over the past two years. First, the object-oriented design of TSA allows us to give new users a quick overview of its structure. This is important since a user needs a working model of a computer system in order to make efficient use of it. The main ideas behind OOP (objects and message passing) are easily explained to new users.

Second, the nature of interactive time series analysis requires that analysts be able to implement new ideas within TSA easily. There are several levels of changes that need to be supported. On the lowest level, certain "on the fly" changes are easy to make because of the persistence of objects in the Lisp Machine. This type of modification is usually accomplished by evaluating a Lisp expression that causes one or more existing objects to be manipulated by sending already defined messages to them. On a slightly higher level, more substantial ideas can be implemented, first, by defining new methods (or modifying old ones) to handle the specific analysis problem at hand and, second, by evaluating Lisp expressions to evoke these methods on existing objects. Finally, more fundamental changes to TSA can be made using the user extensibility features that are discussed in Section 5. These latter two levels of changes are supported by features in OOP such as inheritance and method combination (since these encourage incremental changes to existing software).

A third reason for using OOP that is now apparent to us is that the conceptual decomposition of TSA into objects has facilitated the organization of its software. This has been particularly important since we have made some rather radical changes to portions of TSA over the past two years as our ideas about how best to construct various parts of it have been refined (usually from insights gained from demonstrations to colleagues and from observing new users). Because OOP forced us to associate code with particular objects, we have been able to make radical changes in one portion of TSA without worrying unduly about their effects on other portions.

4. Design of TSA

Interactive time series analysis typically involves the manipulation of a time series with a signal processing or statistical algorithm. The result of this manipulation can often be summarized by one or more graphs. A prime example would be computing the discrete Fourier transform of the series and plotting the magnitude squared of the transform (when properly scaled, this is called the periodogram of the time series). Our design of TSA in terms of OOP thus makes use of three main types of objects (called flavors in the Symbolics OOP implementation): a data object, which contains the time series and certain quantities related to it; frame objects, each of which corresponds to a particular algorithm for manipulating a time series (the use of the word "frame" here is a historical remnant of an earlier version of TSA and is unfortunate due to its definition in Lisp as a generalized property list); and graph objects, which contain information necessary to plot the results of a particular algorithm. We describe these flavors (classes of objects) in more detail and then proceed to discuss how they are tied together in TSA.

4.1 Data Object

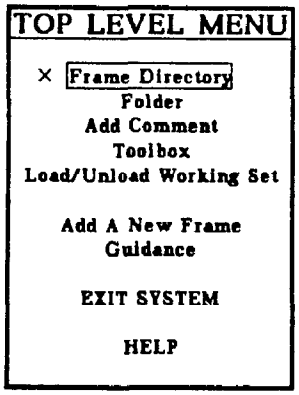
The flavor **data-object** contains instance variables (slots) that specify the properties of a particular time series. These instance variables are (with one exception) set once and for all when a particular data object is instantiated. Examples of instance variables are: the actual values of the time series; the symbolic units for the time series values; and the network name of an ASCII file from which the values of the time series were read. If we ignore the one exception, instances of data objects in TSA are inviolate objects once they have been created.

The one exception is an instance variable called *stat-facts*. Its value is an association list that is used to save various quantities related to the time series. Examples are the sample autocovariance function for the time series and a list of autoregessive models that the user has fit to portions of the associated time series (each model in the list is in fact an instance of a model object). The purpose of *stat-facts* is really to get around a problem with Symbolics's old implementation of Flavors. When we first thought about the appropriate instance variables to include in the flavor **data-object**, it became clear that it would be hard to come up with a comprehensive set *a priori*. This presented a problem because the old implementation of Flavors invalidates existing instances of a flavor if its definition is changed to introduce more instance variables.

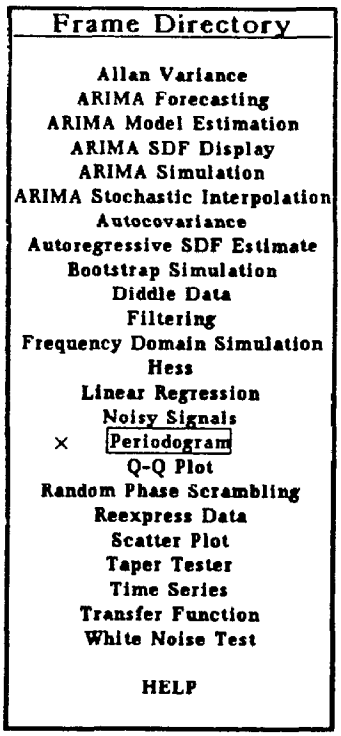
The use of *stat-facts* allows us to simulate the addition of instance variables. For example, suppose that we want effectively to add an instance variable called *sampling-time*. We can implement "get" and "set" methods for it by defining methods **:sampling-time** and **:set-sampling-time** for the flavor **data-object**. The method **:sampling-time** would return either the element

Figure 1

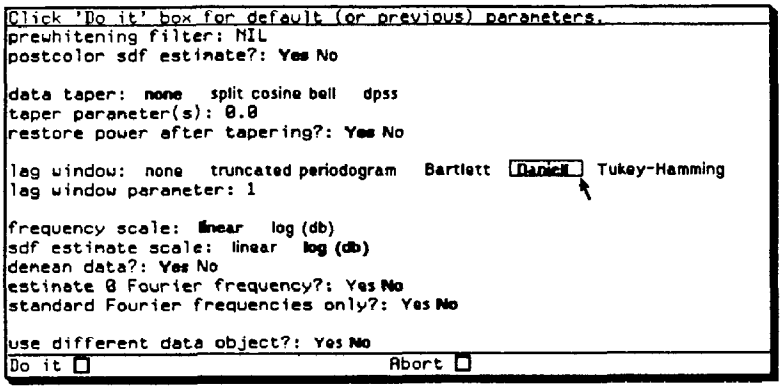
(a)



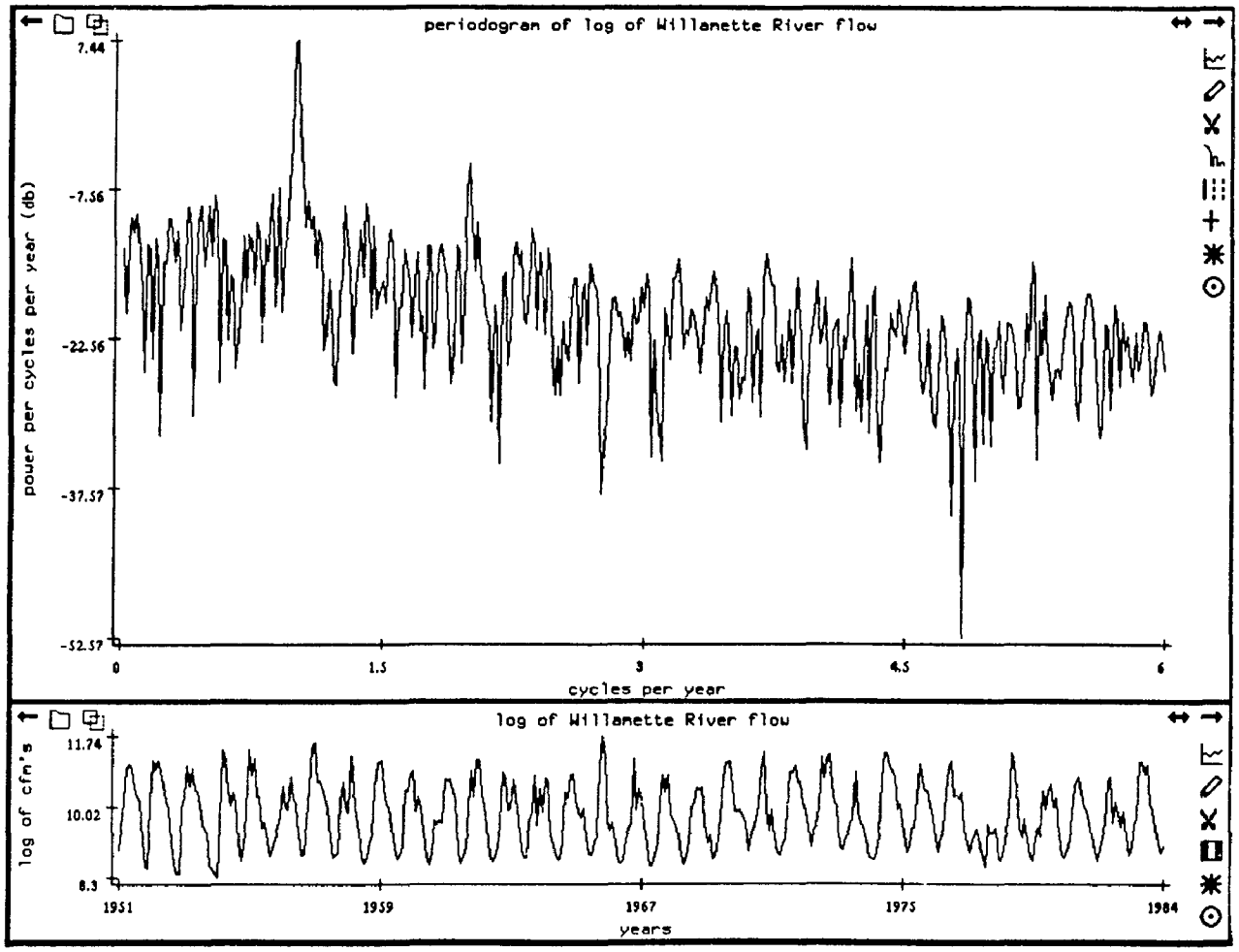
(b)



(c)



(d)



associated with the key-word :sampling-time in the association list pointed to by *stat-facts* or some default value (typically nil) if there is no sublist keyed by :sampling-time. The method **:sampling-time** would require a single argument and would place a sublist in *stat-facts* keyed by :sampling-time and with an associated value specified by the single argument. This scheme allows other portions of TSA to deal with *sampling-time* as if it were a true instance variable of **data-object**.

Another use for *stat-facts* is to cache calculations that are rather expensive to repeat often. An example is calculation of the sample autocovariance function. This quantity is needed in a number of different frames in TSA. It is obtained for a particular instance of **data-object** by sending the instance the message **:acvf**. The method that handles this message looks in *stat-facts* for a sub-list keyed by :acvf. If there is none, it causes calculation of the sample autocovariance function to be carried out and saves the result in *stat-facts* (associated with the key-word :acvf) before returning it to the message sender. A second request for the sample autocovariance function would not require its recomputation.

4.2 Frame Objects

Instances of frame objects in TSA are responsible for retrieving information from instances of **data-object**, causing certain calculations to be carried out, and sending messages to instances of graph objects in order to display the results of these calculations. There are many different flavors of frame objects in TSA (see Figure 1b for a menu of them). For example, an instance of the flavor **periodogram-frame** can be used to make a periodogram-based estimate of the spectrum of a time series associated with a particular instance of **data-object**.

All flavor definitions for frame objects are built upon a component flavor called **generic-frame**. This component flavor contains methods and instance variables for integrating frames into TSA in a uniform fashion. In order to illustrate the role of **generic-frame** in providing this uniformity, it is helpful here to briefly describe how a particular frame is evoked by a user and how a frame responds to particular messages sent to it.

The mouse on the Symbolics Lisp Machine has three buttons (referred to as the L, M, and R buttons). An R mouse button click over an active window that is part of TSA always causes the TSA Top Level Menu to appear (Figure 1a). One of the items in this menu is "Frame Directory." Selection of this menu item causes a second menu to appear (Figure 1b). This is just a directory of all the frames currently in TSA. If we select any one of the items in this menu (say, periodogram), the message **:start-up-the-frame** is send to the value of a global variable (*periodogram-frame* in this case) that points to an instance of the flavor associated with the selected frame (here **periodogram-frame**). This message in turn causes other messages to be sent to the instance of **periodogram-frame**, including **:get-choices-from-user** and **:do-your-thing**. The former results in the exposure of a user-query menu that allows the user to specify exactly how the spectrum is to be estimated (see Figure 1c), after which the latter message causes the periodogram-based estimate of the spectrum to be calculated and plots of the estimated spectrum and of the corresponding time series to be drawn on the screen (Figure 1d).

The flavor **generic-frame** and its associated methods are designed to handle certain messy details common to all frames. For example, the message **:start-up-the-frame** is defined for the **generic-frame** and handles deexposure of graphs, selection of the particular instance of **data-object** to work with, etc. Flavors such as **periodogram-frame** that are built upon **generic-frame** need only supply two things: instance variables that specify the way an analysis is to be carried out (these usually correspond to variables that can be set in a user-query menu such as the one shown in Figure 1c); and methods to handle the particular analysis that the frame is designed to do. For the flavor **periodogram-frame**, this means defining instance variables for all of the options shown in Figure 1c and methods to handle the messages **:get-choices-from-user** and **:do-your-thing**. The first method supports the query-user menu, whereas the second provides an interface to a set of Fortran routines that do the required numerical computations.

4.3 Graph Objects

Instances of graph objects in TSA are used to produce graphical output from calculations controlled by the various frames. Two examples of this output are shown in the two panes of Figure 1d. The upper pane is a plot of a periodogram-based estimate of the spectrum of the associated time series, which is plotted in the bottom pane.

The definition of the flavors associated with graphs is much more complex than those associated with frames. Whereas specific frame flavors are built upon the single flavor **generic-frame**, graph flavors are typically built upon a number of different flavors. We have attempted to keep the inheritance structure rather simple. Basic plotting capabilities (such as axis drawing, labelling the plot, etc.) are provided by the flavor **basic-graph**. The flavor **log-linear-graph** is built upon **basic-graph** and handles switching the scaling back and forth between log and linear. The flavor **dynamic-graph** is built upon **log-linear-graph** and supports the ability to rescale a graph by pointing with the mouse. In a similar fashion other flavors are defined one at a time until we get to the flavor

time-series-graph. This flavor supports a comprehensive set of graph operations and is the one upon which many other graph flavors are built.

When a new type of graph is needed in TSA (see Section 5), a new graph flavor is usually defined by finding an existing flavor that is as close as possible to the desired new one. The new flavor is built upon the existing one and typically extends its capabilities by defining new instance variables and new methods and/or modifies the existing flavor by redefining some of its methods.

4.4 Main Control Loop and the Mouse Process

The heart of TSA consists of a main control loop that looks for blips coming into a control buffer. All of the windows associated with TSA share this same control buffer. There is also a separate process associated with the mouse. When a mouse button is clicked, a blip is forced into the TSA control buffer, and the main control loop dispatches this blip based upon what the click was and where the mouse was when the click was made. The dispatching of this blip is uniform and takes advantage of the ability in OOP of a single message being interpreted in different ways by different objects. Thus, for L and M mouse clicks, the main control loop simply dispatches the message **:handle-blip** to the appropriate object that the mouse was over at the time (an R click is reserved for popping up the TSA top level menu). This uniformity is a clear advantage in keeping the control structure of TSA as simple as possible.

5. Incorporation of User Extensibility into TSA

Extensibility in the TSA system is focused on adding additional frames along with their associated graphs and data manipulation operations. In this section we discuss the facilities in TSA that aid in extensibility and how the object-oriented structure of the system has made the incremental development of these facilities possible. The development of these facilities is being done in two stages. The first was to provide utilities for a programmer with some familiarity with Lisp and the structure of TSA. The second stage, which we are currently involved in, is to provide a higher level interface to the extension facilities so that users with only a casual understanding can effectively add and customize their own frames and graphs. We don't wish to claim that this work solves many of the general problems of automatic programming from high level specifications. For example, there are many natural constraints on a reasonable extension to the TSA system that would not hold for other problems.

We assume that the majority of users interested in extensibility have a Fortran 77 subroutine which they understand well enough to adequately describe the input and output parameters. The interface from TSA is constructed with respect to this subroutine, although the subroutine may itself call other Fortran subroutines and functions.

The steps which involve user interaction in installing a new frame are:

[1] Name the new frame (hopefully based on the intended data transformation), and initialize construction of a new source file for it.

[2] Give the pathname of the Fortran subroutine and provide a description of the parameters in the declaration line.

[3] Describe pre-frame-invocation events (i.e. user-settable runtime parameters) and post-invocation events other than graph plotting.

[4] Select single or multiple graphs which are closest to the graphs the new frame will need from a library of graph types already known to TSA. This includes defining special titles, scales, units, etc.

[5] Add specialized methods for operating in a meaningful manner on the displayed data if it needs more than the standard operations provided for all graphs.

[6] Hook [1]-[5] together and choose whether to make the new frame and graphs a temporary or permanent part of the system.

The original implementation of frame extensibility was done in a monolithic manner that only tangentially utilized the object decomposition of the system. As might be expected, if the user made a mistake the errors quickly cascaded, and recovery by the naive user was improbable. While modifications to existing frame and graph classes were made relatively easily, hooking all of the necessary operations together became very messy. A second effort focused on the addition of "meta-level" objects which helped to keep track of the state of construction of new classes of TSA objects, i.e. a new frame, the graph for the frame, the invocation of message passing from the menu level, etc. This was accompanied by read-along text to provide the naive user with a description of what was going on. Operations were added to monitor progress in constructing the new frame, recover from errors, and restart the process at any one of four particular phases. Generic graphs were used, and any radical customization required competent Lisp programming skills. Suites of related operations were still bundled together, and there was no unified high-level interface. However, the user could couple in a new Fortran routine and get a reasonable graphic display of the output with only a modest level of understanding of TSA.

We are now completing a final (hopefully) extensibility package, in which all operations are done through

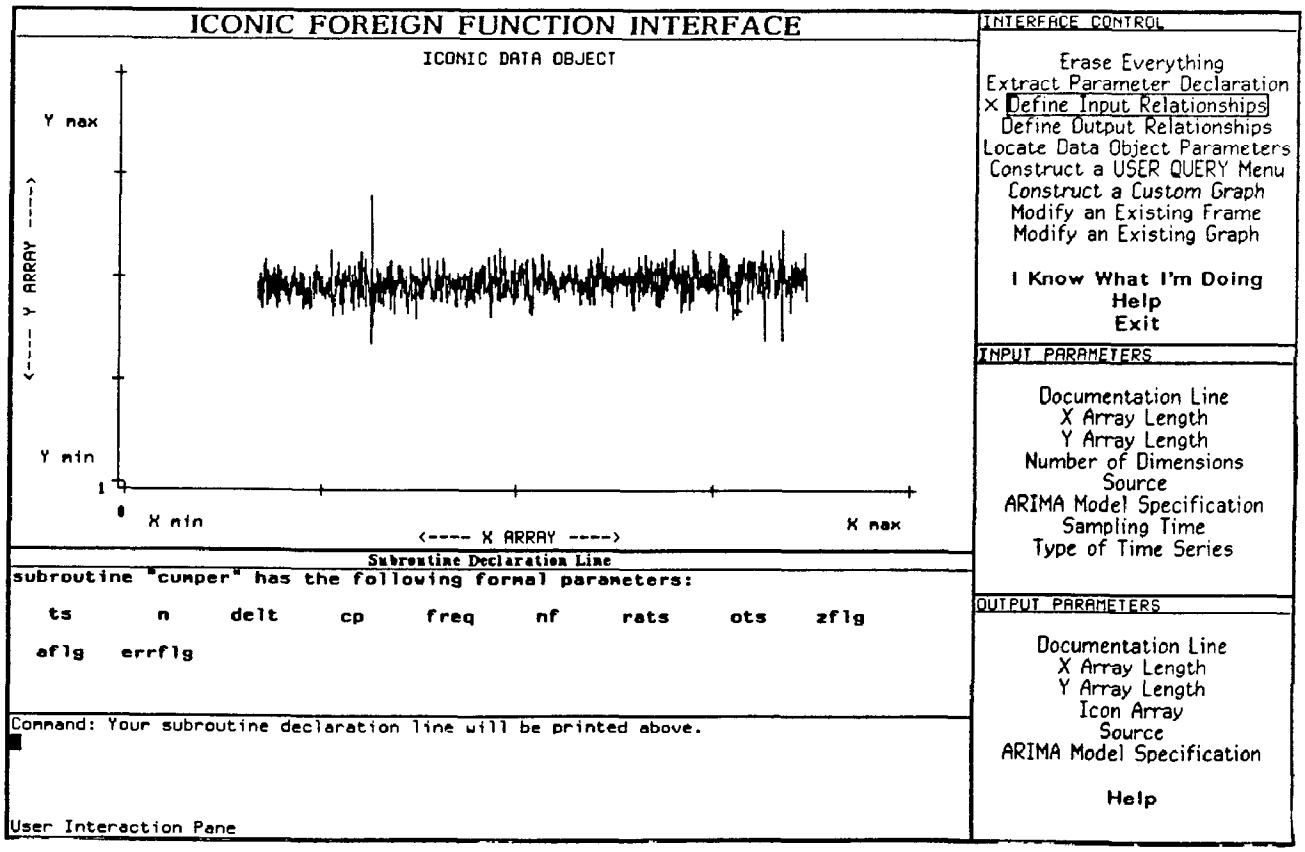
graphical interfaces which connect screen representations of data objects, subroutine parameter lines, graph plots, and other objects of the TSA system with their underlying class templates. The description of steps [1] through [6] above will be mainly with reference to these interfaces.

When the user selects "Add A New Frame" from the TSA top level menu (see Figure 1a), the Iconic Foreign Function Interface appears in a form closely resembling Figure 2 (at this time the bottom two panes in the left column contain different information). Until the end of step [3] the user interacts only with this interface. The Iconic Foreign Function Interface graphically consists of a left hand column for displays and user type-in and a right hand column with various menus that control the current operations of the interfacing process. The top pane in the left column contains an iconic representation of a generic input data object. All of the relevant attributes are mouse-sensitive, so a user can click on any of them when indicating how each possibly corresponds to some input parameter of his or her Fortran routine, the declaration line of which appears below in the Subroutine Declaration Line window. At the bottom of the left column is a User Interaction Pane which is used for prompts as well as user input. In the final version of the interface this pane may have Lisp evaluation capability along with user-directed mouse-sensitivity.

The right hand column contains menus which complement normal control flow in adding a new frame or augmenting an existing one. For example, if the menu item "I know what I'm doing" is highlighted in the Interface Control Menu, many of the prompts are not given, and the user may control the selection and ordering of steps in the addition process. This frequently happens when an experienced statistician using TSA for exploratory programming discovers a new display format or a new input or output parameter that would be useful.

In step [1] the system queries the user to obtain the new frame name and documentation comments. Once these are obtained, they are substituted in the appropriate places in a text template that contains the skeletal structure common to all frames. Although many Lisp implementations offer elegant "grinder" functions which allow a user to generate a textual representation from internal Lisp forms, we found problems using these because the syntax of Flavors differs from that of the embedding language, Zetalisp. When object-oriented features are added to an existing language, it seems much more reasonable to preserve a uniform syntax. Hopefully, this will be done in the proposed standardized object-oriented extensions to Common Lisp (Bobrow *et al.* (1986)).

Figure 2



The next two steps serve to integrate communication between the Fortran and Lisp worlds, which coexist in the large linear address space. In the Symbolics implementation of Fortran (see reference [there are several special constructs which allow both the Lisp world and the Fortran world to share data space. Inputs to most Fortran subroutines used in statistics generally consist of one or more numerical arrays along with runtime-settable scalar "flags" that control inter-subroutine operations. Outputs are the transformed arrays along with scalars indicating error codes or other descriptors of the transformation. Lisp macros provide a uniform way of describing storage allocation in the shared data space, and they are also used for passing the actual parameters when the subroutine is invoked.

In step [2] TSA queries the user for the pathname of the Fortran subroutine, reads the source file, and looks for the subroutine declaration line. The declaration line is displayed in the appropriate window, and the formal parameters are made mouse-sensitive, making it possible to connect them with their actual counterparts in the particular instance of **data-object** that will be used for input at runtime (henceforth called the Current Data Object). Figure 2 depicts this point in the frame addition process. Many of the actual-to-formal parameter correspondences can be mapped between the iconic representation of the Current Data Object and the Fortran declaration line, but those that have no iconic counterpart must be moused on in the Input Parameters menu. User prompts appear when necessary to complete this process. For much of this specification process it is possible just to deal with the iconic representation of the parameter "objects". Attributes are stored with each parameter object so that later it is possible to debug or query the new frame regarding the correspondences. This use of objects and their iconic representation spares the statistical analyst from the intricacies of low-level interface programming.

In addition to parameters that are actually part of the data to be transformed, there are frequently parameter "flags" that indicate to the Fortran routine that the normal operation of the routine should be modified. In typical Fortran programs these flags are just integers, and it is therefore easy to get confused about what conditions are in effect at runtime. Step [3] remedies this by allowing the person adding the new routine to specify that a user-query menu with a selection of all possible runtime modifications be presented so that a later user of the new addition will be aware of not only what modifications are available, but also what their implications might be for the transformed data. It is possible to construct simple versions of this runtime menu now, again by starting with an appropriate menu object as the class template. Short English descriptions of the modification are typed in the Interaction Pane, and TSA uses these to write out a specific menu for the new frame. (Figure 1c shows the runtime choices for Periodogram Frame.)

At this point (end of step [3]) the frame file for the new routine is almost ready to be written, compiled, and included in the TSA system. However, the new frame has only taken care of getting the data from the Current Data object to the Fortran routine. Typically it is desired that one or more graphs be displayed that summarize the calculations done by the Fortran routine. To accomplish this, the person adding the new routine must describe one or more appropriate graphic displays. The most important structural concept in extending the TSA system has been the ability to use and modify existing flavors to produce a flavor that specifies a new type of object. This procedure is also used to construct new types of graphs in step [4].

In order to achieve this, TSA keeps a library of all its graph types. When it comes time to "Describe a Custom Graph," a pop-up menu appears with all types listed. The person adding the new routine may click through them selecting the one that is closest to what he or she has in mind. As the number of graph types known to TSA grows this approach may become tedious. We will need to devise a better system of selecting a starting point for customization. A basic "time series graph type" is available as a default if the user just wants a quick and uncomplicated display.

Once the nearest graph type is selected, many of its components are made mouse-sensitive and it is iconically displayed. Titles, scaling factors, etc. are directly typed on the iconic representation. Then the output parameters (results) of the transformation are connected to the graph type in the same way that the input parameters were connected to the iconic representation of the Current Data Object. In many cases more than one graph is needed, and we assume that the output arrays contain the required numerical data for all displays. In the case that a statistician is "exploring" with a new routine, he or she may wish to add another graph and display a slightly modified form of the data which is already being displayed in previous graphs he or she has chosen. Presently this demands that either a little Lisp hacking or recoding part of the Fortran subroutine be done.

For most graphs many of the operations icons at the top and right hand side of the graphical displays are appropriate. Most likely, however, the new addition will require some graph operations which are unique to the particular type of transformation that is being displayed. Thus far we have not tried to add any of these using just the high-level interface. There is a great deal of similarity in the structure of these methods, and it is possible that we will discover a general scheme which permits us to use a generic method type and allow the user to modify it correctly to a newly created graph operation. We believe

this will work for many displays, but not for all. The only part of this section (step [5]) that works automatically now allows to user to go into the icon editor and design the appropriate icon for the graph operation. This is then made mouse-sensitive and becomes a part of the new graphic display.

After all the various graphs and methods needed for the new routine are described, the user then blocks out the display layout with a few mouse clicks. The graphs are created and displayed (without data) just to be sure that everything fits on the screen. If the person making the addition is not satisfied, he or she iterates through the graph design section until pleased with the results. The new frame and its graphs are now fully described. All source code not yet written is constructed (step [6]), the files are compiled and relinked (this takes little time with the Lisp machine's dynamic linking), and the new routine is ready for testing.

At each refinement of the extensibility module both the capability and organization were dramatically improved as we relied more heavily on the object-oriented structure of both TSA and the Symbolics environment. On one level we could add new classes in the inheritance lattice based on individual refinement of existing ones. On another level we could construct meta-level machinery to carry out this refinement process while also integrating it with existing Symbolics tools like the flavor examiner (browser). The uniformity of the object-oriented paradigm has added a level of consistency and reduced the apparent complexity in a large and continually evolving system.

6. Evaluation of the Object-Oriented Approach

We conclude this description of our use of an object-oriented methodology in constructing the TSA system with a few brief remarks on the suitability of the paradigm in the context of time series analysis. Some of these observations mirror a slight discomfort with everything being dealt with as objects (although we freely made the choice to do so), and others strongly support the use of that approach. There are many theories in cognitive psychology regarding the refinement of conceptual structures as one's understanding of a subject progresses from naive to expert. We believe that the object structure helped greatly in providing a bridge between various levels of conceptual modeling. It allowed both naive and experienced statisticians to quickly understand the TSA system structure, yet rearrange the possibilities of its use to fit the level of statistical sophistication. Beginners could follow along a prescribed analysis path, asking for help when needed; yet experts could quickly flip to selected parts of the system to make modifications suggested by exploratory data analysis. We think the modular structure of objects and message passing helped in providing that flexibility.

Another strong feature that was provided by the inheritance structure of OOP was that as we tested, augmented, and revised various modules of the system we could distill out a great deal of commonality that was not apparent in earlier implementations. We could then go back and encapsulate more primitive ideas in more basic classes (flavors) and thus reduce the level of complexity visible in the most advanced classes. We realize that this programming technique doesn't fit with "top down design" practices, but in our case we were trying to build a system from a conceptual model of time series analysis, not a formal program design specification. Since TSA is focused, and therefore purposely constrained, to operational data transformations and graphical representation, inheritance also played a key role in the high-level extensibility of the system. Much of the basic functionality of transformational or graphical classes could be provided to a person modifying the system, allowing them to only be concerned with tailoring things for their specific operation (usually with high-level views of previously constructed classes and methods which implicitly suggested new possibilities).

On the other hand, there are several situations in which a more procedural interpretation of data manipulation seems to be more natural. That conceptual model is no doubt influenced by historical Fortran roots, but it does emphasize the fact that language paradigms should not be taken as absolute specifications. We must continually examine both old and new paradigms, and use them when it seems appropriate. An example of this occurred in the incorporation of "access-oriented" ideas, where evaluation was computationally costly. No evaluation takes place, unless a message passer specifically requests it.

We also found a small but potentially dangerous inconvenience associated with the incorporation of both encapsulation and data abstraction together with inheritance. In a more basic class (flavor), the implementation of instance variables is hidden from the outside. But what is the status of this implementation when the basic class is incorporated as a component in a more complex class? If a method in the more complex class needs to access an instance variable of the more basic class we have relied on knowing about that implementation. For instance, in the case of an instance variable in the basic class being implemented as an array, we access it with arefs and asets (the access functions for arrays in Zetalisp). On the one hand we argue that this is all right because the natural mental model of a statistician is that it *should* be an array, and that hiding this would necessitate writing seemingly superfluous access methods in the basic class. On the other hand, the benefits of information hiding break down if for some reason we later decide to implement the instance variable in the basic class as a list instead of an

array. Generic access functions like the setf of Common Lisp may help, but we think the problem is more complex than this.

All in all, though, we are very pleased with the object-oriented framework of the TSA system. It provided not only guidelines for good programming style and maintainability as the system grew, but it also helped us to clarify a conceptual model of data analysis that we have found suitable for a large range of users.

7. Acknowledgement

This work was funded by the Office of Naval Research.

8. References

[1] Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. (1986), "CommonLoops: Merging Lisp and Object-Oriented Programming," in *OOPSLA '86 Conference Proceedings* (special issue of *SIGPLAN Notices*, **21**(11)), edited by N. Meyrowitz, Association for Computing Machinery, pp. 17-29.

[2] McDonald, J. A., and Pedersen, J. O. (1985a), "Computing Environments for Data Analysis, Part I. Introduction," *SIAM J. Scientific and Statistical Computing*, **6**(4), pp. 1004-1012.

[3] McDonald, J. A., and Pedersen, J. O. (1985b), "Computing Environments for Data Analysis, Part II. Hardware," *SIAM J. Scientific and Statistical Computing*, **6**(4), pp. 1013-1021.

[4] McDonald, J. A., and Pedersen, J. O. (1986), "Computing Environments for Data Analysis, Part III. Programming Environments," Technical Report No. 82, Department of Statistics, University of Washington, Seattle, WA.

[5] Tukey, J. W. (1984), "Data Analysis: History and Prospects," in *Statistics: An Appraisal*, edited by H. A. David and H. T. David, Iowa State University Press, pp. 183-202.

[6] *User's Guide to the FORTRAN 77 Tool Kit* (March 1985), Symbolics Corporation, Cambridge, MA.