Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

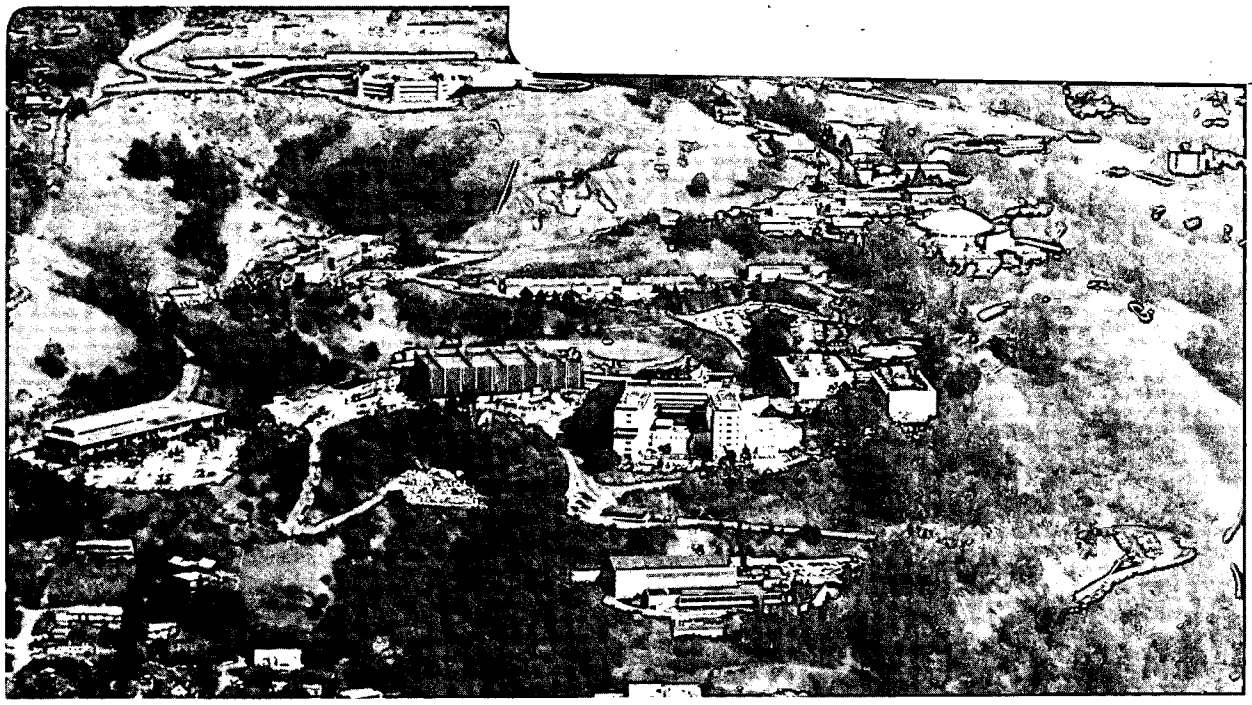
Engineering Division

Presented at the Workshop on Model-Based Control Software, BNL, Upton, NY, August 17–18, 1987, and to be published in the Proceedings

Towards Portability in Model-Based Control Software

V. Paxson and E. Theil

November 1987



Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098

DISCLAIMER

Towards Portability in Model-Based Control Software

V. Paxson and E. Theil

Real Time Systems Section
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

November 1987

This work was supported by the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

Towards Portability in Model-Based Control Software

V. Paxson
E. Theil

Real Time Systems Section
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

November 2, 1987

Introduction

One of the cardinal rules in developing any software system is "Don't reinvent the wheel." Simply put, this means that whenever possible reuse existing codes rather than writing new ones. The reasoning behind this rule is straight-forward: abiding by it can save an enormous amount of labor which can be better spent concentrating on those aspects of the system which are truly new.

In general, the degree of ease in reusing existing software is greatly influenced by the *design* of the software — what assumptions it makes about the data it operates on and the types of output it should generate. Modeling programs (e.g., MAD, COMFORT, TRANSPORT, TEAPOT, ...) are usually easy to reuse because they make few assumptions about their input data other than that it will be read from a file in a prescribed format. They generate simple forms of output such as ASCII files, and they are written in languages such as FORTRAN-77 which are transportable across a wide spectrum of computers.

What has proven much less easy to reuse are the large bodies of software which constitute the high-level control system of an accelerator. The barriers to reuse have been that the software often has an intimate knowledge of the accelerator it controls (assumptions about the input data) and the display hardware such as graphics devices to be used for output (assumptions about the type of output). Because of these problems, little control software gets reused, and much labor is spent reimplementing similar software packages for different control systems.

A couple of recent developments [1] offer the possibility of change for the better:

- **Model-based control software,** i.e., software designed around one or more *models* of the control process, does not have the problem of assumptions about the input data. As long as the input data can be plugged into the model, the software can operate on it, because it does not have the particulars of a control system wired into it.
- **Emerging graphics standards** offer a way to design software such that the assumptions made about the output to generate will remain valid when the software is moved from one type of computer to another. We take "graphics" to mean not only the now-familiar pictorial displays like data plots but window systems, such as those available on scientific workstations, as well.

In light of these developments, the crucial part of making accelerator control software *portable*, i.e., reusable on different accelerator control systems and on different computers, remains the software's *design* — the basic structure of the program. In particular, given a sound design, it is possible to change how the software acquires the data for its model, and how it does its graphical output, with *no* change necessary to the main body of the code. On the output side of things, this extends to being able to adapt the software to a different, non-standard graphics environment with minimal effort.

What is Model-Based Control Software?

When used today the term "model-based control software" generally refers to control software built on top of one of the modeling programs mentioned above.

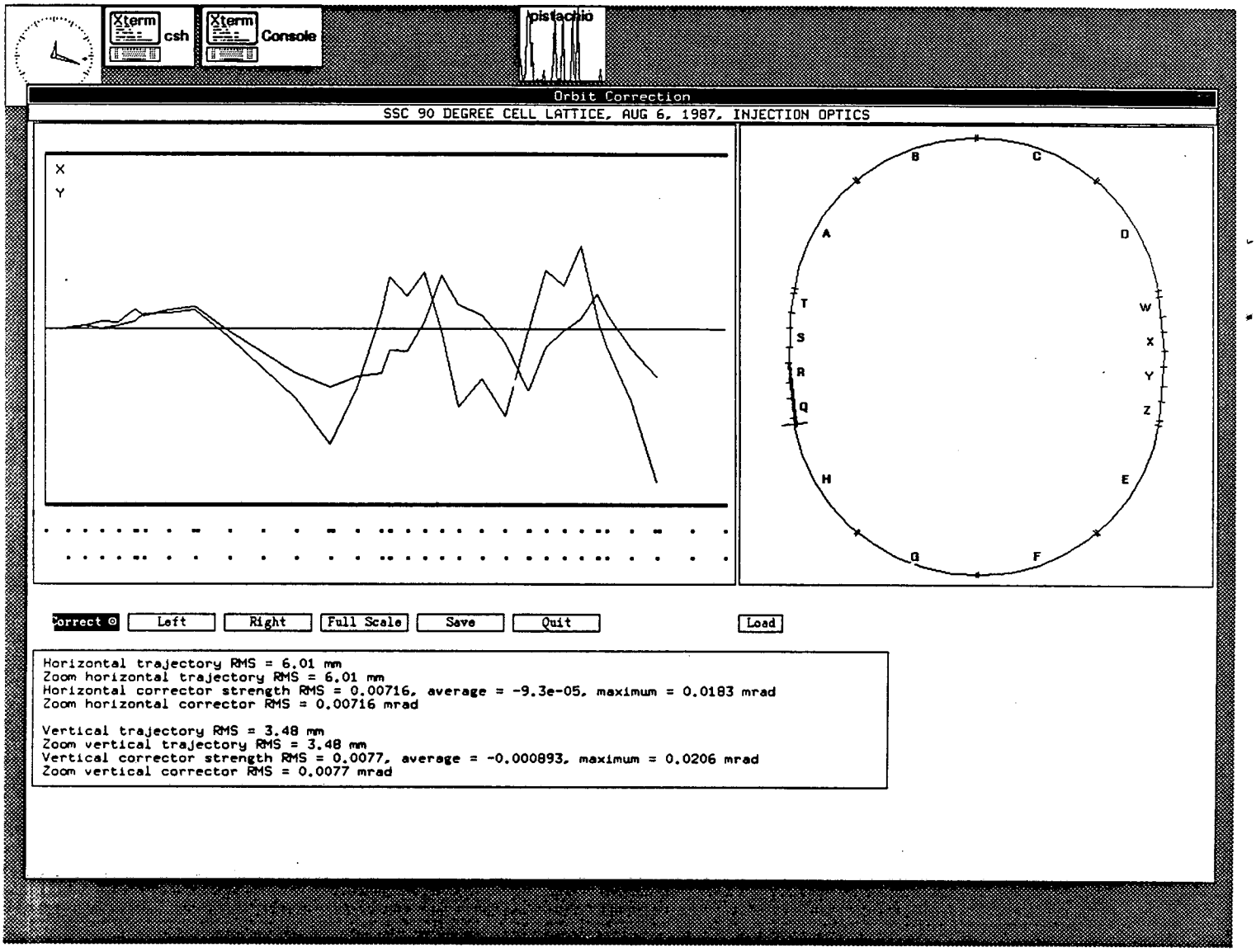


Fig. 1: **Orbit Correction**, a fairly typical model-based control program with a graphical interface. The user designates regions of the accelerator to correct and monitors the orbit correction algorithm's progress as it attempts to improve the trajectory.

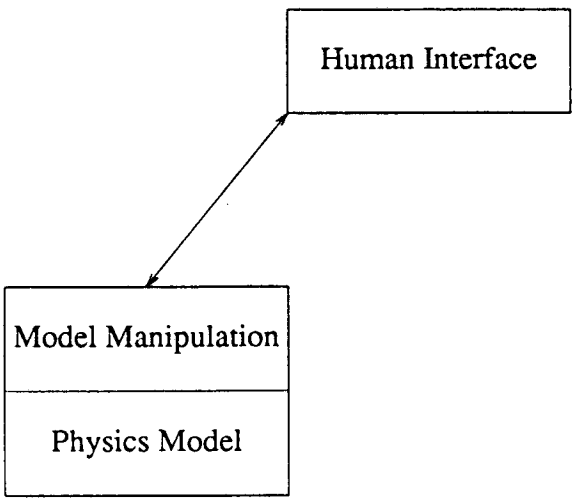


Fig. 2: Conceptual structure of programs like **Orbit Correction** which are based on a model of the underlying accelerator physics. The main part of the program occupies the *Model Manipulation* box.

A typical example is shown in Fig. 1. Orbit Correction has two displays: a top view of the entire machine and a "zoomed-in" current region of interest, in which the X and Y beam readings are plotted. The user moves and expands the region of interest until it matches the part of the machine that needs correction, and then invokes an orbit-correction algorithm to flatten the beam in the region. Orbit Correction is built on top of TEAPOT [2], though it is not tightly coupled with TEAPOT (it runs as a separate process), a point we shall return to later.

In general, these kinds of model-based control programs have a structure like that shown in Fig. 2. The core of these programs, that is, the part within them which knows the physics necessary to affect the accelerator in a particular way, is the box labeled "Model Manipulation". This core is built on top of the model of the machine's underlying physics.

One important point is that the programs do not need the *complete* physics model; they operate on a *partial view* of the model. Orbit Correction needs a list of the machine's monitors and correctors and their position around the ring, and the machine functions. It does not need any knowledge about the underlying physics of why the given lattice has its particular set of machine functions, it only needs to have some way of finding out the results of a set of changes to the corrector strengths. As shall be discussed later, that these types of programs have *partial* and not total views of the model makes them much easier to port.

The control programs get their input from and write their output to what are often quite complicated human interfaces, which can involve presenting information to the user in a highly graphical form, and in such a way that the user can manipulate these pictures directly. For example, to designate an area of interest when using Orbit Correction, rather than typing in the names of the first and last elements, the user points to them on either the top-view or the current region of interest, zooming in or out as needed. In the figure, this portion of the control program is labeled "Human Interface".

The presence of a rich interface is a two-edged sword: while it makes the use of the control program much more natural and efficient, it has also been one of the major stumbling blocks in attempting to reuse codes on different computers, since graphics and particularly window systems have until recently been unique to each computer manufacturer. We shall address this point later, too.

Other Models

While the term "model-based control software" usually refers to programs like those just described, in which the underlying model is the physics of the accelerator, there are other equally important, non-physics models used in accelerator control systems, and programs built on top of these also fit well into the concept of "model-based".

Fig. 3 shows "On/Off Control", a program used at the Bevatron to turn beamlines on or off. The display shows a picture of the beamline switchyard. The user can select an individual magnet, compose a group of magnets, or select an entire beamline, and then control it from the menu. The *elements* of On/Off's model are simple: it knows about magnets, beamlines, and how to turn objects on and off, and that's about it — there is no physics in the model. But the *relationships* within the model are more complicated than those usually found in a physics-based model: On/Off knows a lot about the spatial location of the objects in its model and a lot about the geometry relating one object's position to another's.

Other particular examples are the vacuum subsystem and the safety interlocks, as well as a host of possible CAD-derived applications. These sorts of control programs can be added to our previous conceptual structure as shown in Fig. 4. Like the physics-based modeling programs, these too will have rich human interfaces. The two types of modeling programs can also be connected at their common root, the box labeled "Machine Interface". This typically would be a database which stores within it the accelerator's static and dynamic state. Since the control programs are model-based, the bottom-end can be detached and the upper part of the system used for simulation purposes as well.

Achieving Portability

Given now the nature of the software which we are interested in moving from one environment to another, what are the barriers to being able to load the program into a new computer, compiling it, and having it work? Taking a look at Fig. 4, the problems are encapsulated by the boxes on either side of the "Model Manipulators". When reusing model-based software the modeling program on which it is based may be different from the one used in the new environment,

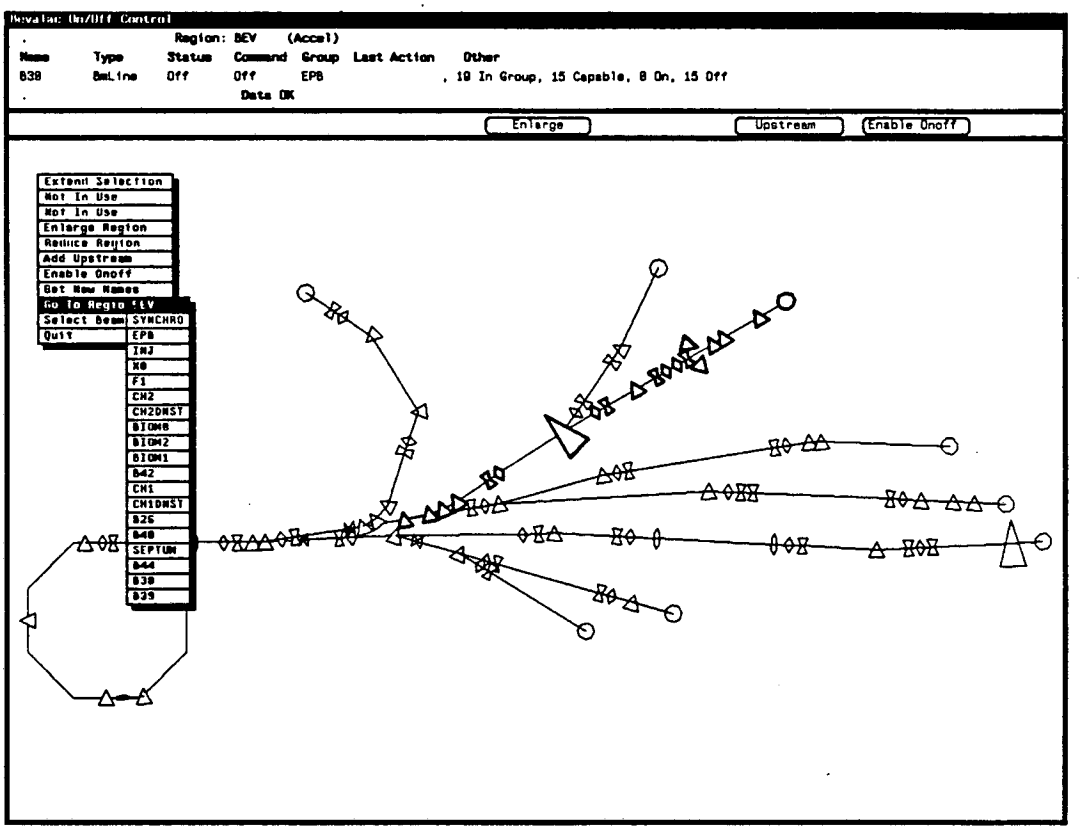


Fig. 3: **On/Off Control**, an example of a model-based control program for which the underlying model is something other than the accelerator physics. **On/Off** knows about the spatial layout of the accelerator and the groupings of magnets and their power supplies.

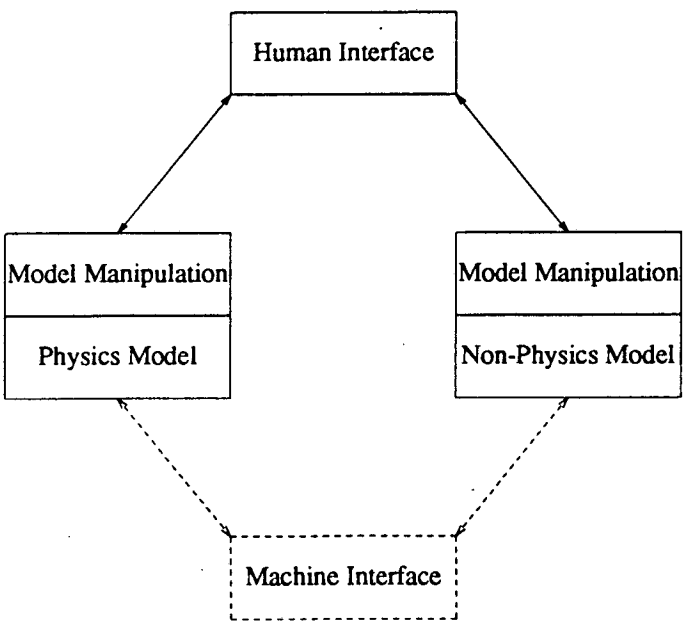


Fig. 4: General conceptual structure of model-based control programs, including both those based on models of the machine physics and those based on other models. Again, the main part of the programs occupy the *Model Manipulation* boxes.

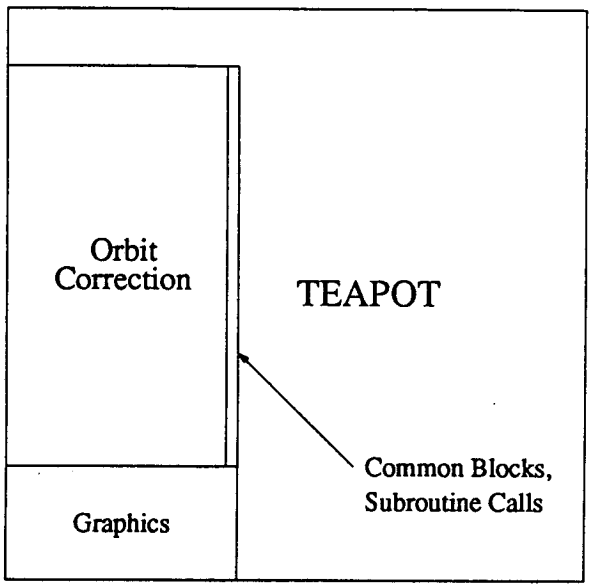


Fig. 5: One obvious way to design the **Orbit Correction** program - make it part of the modeling program, TEAPOT. Due to the tight coupling between the two programs, this design makes portability difficult.

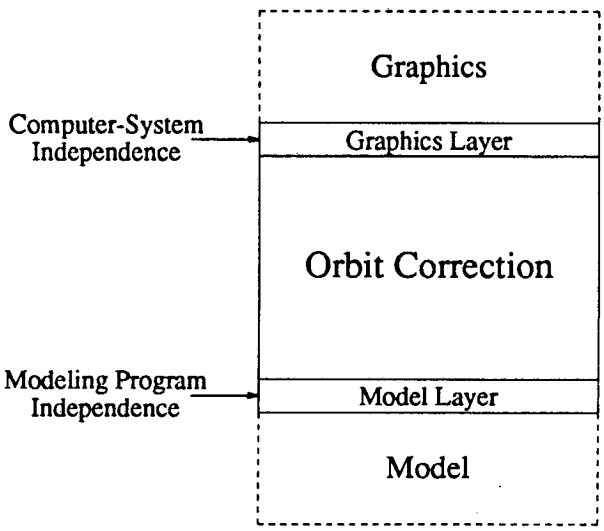


Fig. 6: Alternative design for **Orbit Correction** which stresses limiting the program's coupling to external data and routines as much as possible. This design proves much more conducive to portability.

and/or the graphics and window system comprising the human interface may be different. Both of these barriers usually require a great amount of effort to remove — often more effort than being able to reuse the software is worth.

As stated earlier, the key to making model-based programs easy to transport lies in the basic *design* of the software. To illustrate, consider the Orbit Correction tool mentioned above and shown in Fig. 1. If we were writing such a program from scratch, an obvious way to design it would be to make it part of our modeling program, a feature which would be invoked given a certain argument in the input deck. Such a design is illustrated in Fig. 5. With this design, Orbit Correction is thoroughly integrated into the modeling program. It shares global data with the remainder of the program via common blocks, and it is called by the modeling program when time for it to do its task and it in turn calls routines in the modeling program to do various tasks. The human interface is built on top of a graphics system, which might be a standard one such as GKS or, likely as not, a home-grown one.

From a portability viewpoint the problem with this design is that the task "Orbit Correction" has too much knowledge about the environment around it. It has a strong idea of both the modeling program's data structures and routine calling sequences, and of the particulars of the graphics system, such as how to initialize it, make drawings, and gather input from the graphics devices. To detach it from either the modeling program or the graphics system will be a very difficult, tedious chore, because to do so one needs to *understand* what the various routine calls do and what the data structures mean in order to replace them; i.e., one must have a working knowledge of the very software system which one has no intention of using!

To make Orbit Correction easily portable, it needs a different design. Fig. 6 shows a different way of structuring the program such that it becomes much more self-contained. Conceptually, the "guts" of the program — that part which embodies the algorithms for doing the task "correct the orbit" — occupy the center box. This part of the program is isolated from the rest of the world by two layers.

The bottom layer, called the "Model Layer", is: (1) a set of routines which the program uses to get the data it needs in order to do its task (for Orbit Correction's algorithm, this is the number of monitors and correctors in the lattice, their positions, phase, and beta functions; for Orbit Correction's interface, a title associated with the lattice, the beampipe size, the beginning and end points of the various sections of the machine, and the position and bend angle of the dipoles, in order to construct the top-view); (2) a set of routines which it uses to make changes in the model (in this case, a way to set corrector strengths); (3) a set of routines which implement "black box" functionality needed by the program but which is not directly related to its own function (generate a new trajectory, save the new machine state). These routines comprise Orbit Correction's *partial view* into the complete physics model of the accelerator; they define precisely what the program needs in order to do its task, so to transport the program to a system using *any* modeling program, all that must be done is that these routines be written. Writing them tends to be a very straightforward task, mostly consisting of moving data into or out of common blocks.

Thus, the bottom layer gives the program independence from any particular modeling program.

The top layer, called the "Graphics Layer", is: (1) a set of routines which a graphics interface can call to get information it needs for its display (for Orbit Correction, this is the information mentioned in (1) above, plus a way to get the current corrector strengths and monitor readings); (2) a set of routines the graphics interface can call in order to convey changes the user wishes to make (e.g., set the current region of interest, manually adjust a corrector or monitor); (3) a set of routines which implement, for the graphics-interface, "black box" functionality (correct the orbit, generate a trajectory, save the machine state). These routines define the *partial view* which the graphics-interface has into Orbit Correction. Note that what is provided are routines within Orbit Correction which can be called externally, not routines which Orbit Correction needs to have available in order to do its task. This structure provides tremendous flexibility in terms of human interfaces — for example, it is very easy to write a simple keyboard-oriented, ASCII interface in fully portable FORTRAN which will work on any machine Orbit Correction is moved to, yet does not preclude a much richer graphical interface as well.

In summary, the top layer gives the program independence from any particular graphics system. If the program is written according to the FORTRAN standard, it then becomes fully independent of the computer system it is running on.

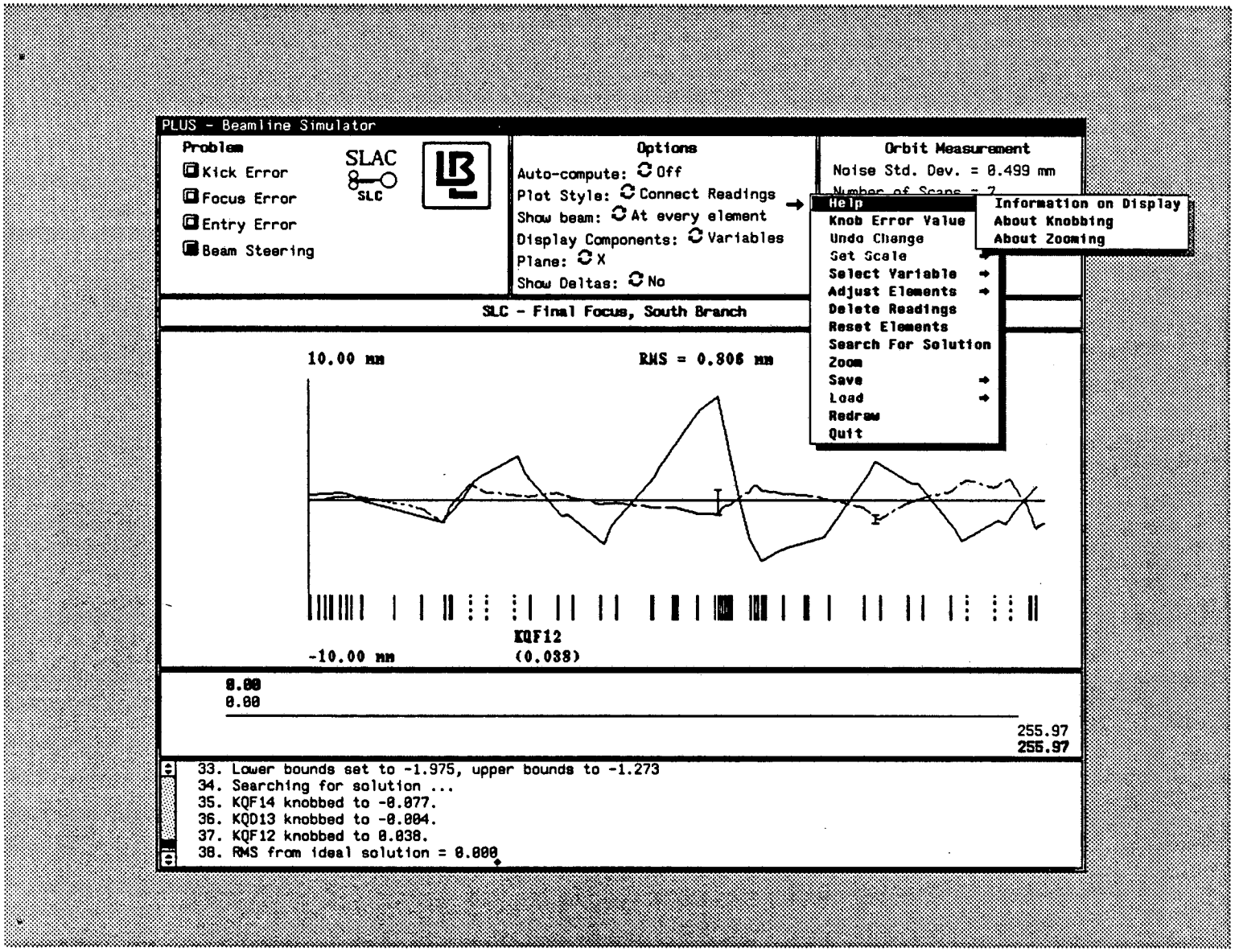


Fig. 7: **PLUS**, a beamline simulator and error-finder which was designed using the structure shown in Fig. 6, and since successfully transported to several different environments.

Experiences

PLUS [3], shown in Fig. 7, is a beamline simulator and error-finder which, in conjunction with colleagues, we designed in accordance with the method outlined above. The first graphical interface was done on a Sun workstation using the SunCore and SunView graphics systems. The latter (SunView) is highly Sun-specific. The complete interface comprised about 5000 lines of C and took two months to develop.

At CERN, we built a second interface which implemented a subset of the first's functionality, on top of Apollo's proprietary Dialog and GPR graphics systems. By using a display library already available, the effort was accomplished in one week, with less than 500 lines of new code being written. No code within PLUS or its "Graphics Layer" had to be modified.

Recently, our colleagues have transported PLUS to the MicroVax environment, again without great effort.

These experiences show that the design structure outlined above does indeed promote portability.

Standard Graphics Systems

The numbers mentioned above — two months and 5000 lines of code for the first interface, one week and 500 for the second — need to be discussed further lest they give inaccurate impressions. The point was not that once the first interface was done it was easy to replicate it; this, in fact, is not the case. While the Apollo interface provides most of the functionality of the Sun interface, the latter is far easier and more intuitive to use, and a much more effective interface. The point was that the program could be detached from 5000 lines of machine-dependent code with extremely little effort.

Truly effective user interfaces can require a great amount of work to design and implement. Because of this, it would be a boon if the graphics interfaces of control software could be transported as easily as the basic functionality of the program.

We are now at a point in the evolution of graphics systems where this ready transportability can be had: enough computer vendors have banded together to produce a standard graphics system, called X-Windows [4], which runs on a wide variety of hardware, including Sun, Apollo, DEC, HP, IBM, and Macintosh computers. X-Windows itself is quite low-level, but designed to support layers built on top of it to provide higher-level graphical operations. Our work at LBL is now being shifted towards using X-Windows (while keeping with the above design, so that the interfaces are separate from the control applications), and we are building packages on top of it to provide easy access to functionality common to accelerator control interfaces (the interface for Orbit Correction is an example).

Summary

Two developments — that of model-based control systems in the accelerator community, and that of the X-Windows standard in the computer graphics community — have made it a realistic possibility that large bodies of high-level accelerator control software can be shared and reused throughout the accelerator community. To take advantage of these possibilities requires the discipline of truly modular software design, in which programs have very well-defined *views* of the data on which they operate and which they make available to the human interface software. If these programs are written with their knowledge of the rest of the control system restricted to simply exactly those things required for the task at hand, they can be easily detached and moved to other environments.

We have presented a general design for producing portable control software, one which has been shown by experience to be effective. We are now using X-Windows so that our labor-intensive user-interface development efforts might prove as readily portable. If enough other members of the high energy physics community join in using the standard, we can overcome the final hurdle to large scale code-sharing.

Acknowledgments

We would like to thank Van Jacobson for many ideas on the design of portable programs, Martin Lee for supplying us with modeling programs and many ideas on the possibilities of model-based control, and Eva Bozoki for similar help in beginning our work.

We would like to thank Martin Lee and Scott Clearwater for our collaborative effort on the development of PLUS.

This work was supported in part by the United States Department of Energy under Contract Numbers DE-AC03–76SF00098.

References

[1] E. Theil, V. Jacobson, V. Paxson, "The Impact of New Computer Technology on Accelerator Control", 1987 IEEE PAC, Washington, D.C.

[2] L. Schachinger, R. Talman, "TEAPOT: A Thin-Element Accelerator Program for Optics and Tracking", *Particle Accelerators*, Vol. 22, pp. 35–56, 1987.

[3] M. Lee, et. al., "Modern Approaches to Accelerator Simulation and On-line Control", 1987 IEEE PAC, Washington, D.C.

[4] R. Scheifler, J. Gettys, "The X Window System", *ACM Transactions on Graphics*, #63, 1986.

LAWRENCE BERKELEY LABORATORY
TECHNICAL INFORMATION DEPARTMENT
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720