Application of a Multipurpose Simulation Design

Jeffery W. Bell[•], Matthew J. O'Rourke[ŧ]
Bihrle Applied Research, Inc.
Hampton, VA 23666

Summary

Seven years ago, Bihrle Applied Research (BAR) began developing high fidelity simulation software on PCs. At the time, the only driving factors were cost and availability. Some inexpensive PCs were available for simulation research, while the better-equipped and more expensive workstations were not. The PCs used in the early research phases were slow and had limited resources, and were generally far less capable than the workstations on which most high fidelity simulations were hosted. This paper discusses the problems BAR encountered developing a high fidelity simulation in a limited PC environment, the decisions that were made to resolve these problems, and the effect some advances in PC technology had on original design goals.

Introduction

Despite the limitations imposed by the PC environment, BAR set ambitious goals in the original design specifications with the assumption that PC hardware and software technology would continue to advance:

1. High fidelity simulation with fully model-independent operation capable of both rapid prototyping and full system simulation.
2. Modular structure allowing easy configuration without programming to support the diverse requirements of the aerospace industry.
3. Integrated simulation development and analysis tools.
4. Object oriented design supporting extensions without code changes to core components.
5. Flight simulator capable of fully modeling an aircraft in real-time on a PC running 32-bit Windows.
6. Reusable tools available to different applications associated with flight simulation, but not necessarily dependent on it.

The term model-independence indicates an application capable of operating with any or no simulation model present. In effect, the simulation model is read from one or more files, and can be changed without altering or restarting the application. As an independent contractor, BAR works with a number of different aircraft models and aerodynamic data sets, including data obtained from short wind tunnel tests targeting the behavior of a specific aircraft configuration in a specific region of flight. Separating common simulation functionality from aircraft-specific behavior was essential to allow aerospace engineers to concentrate on modeling aircraft without becoming software engineers in the process. Additionally, it allowed the simulation developers to concentrate on producing good software without becoming aerospace engineers in the process.

General Structure

One of the first problems encountered early in the design process was the need for including program code in the simulation model. Different types of aircraft with significant differences in flight characteristics, different database mechanization techniques selected by different engineers, the need for a flight control system, and evolving simulation technology all contribute to the complexity of the simulation model. These factors make it nearly impossible to develop a simulation model for use in a high fidelity simulation without writing some minimal amount of code.

The ideal solution would have been an interpreted language built into the simulation application. The simulation application could read all data and code associated with a given simulation model directly. Unfortunately, interpreted languages are, by definition, slower than compiled languages, and performance is always a concern in a PC environment. Additionally, porting simulation models between other simulation platforms would be more difficult if it involved converting between different programming languages. And while neither portability nor language independence was a specific design goal, they are nearly always goals in a good software design as long as they do not compromise other more critical goals.

[•] Software Engineer
[ŧ] Aerospace Engineer

The Windows operating system (OS) provided a relatively simple solution using Dynamic Link Libraries (DLL). A DLL is a library of functions with built-in support for run-time linking. In effect, a DLL can be loaded into an application's address space after the application has been started. Functions exported by the DLL can be imported (called) by the application and functions exported by the application can be imported by the DLL. In reality, the only difference between an application and a DLL is an application begins and ends the process. Both application and DLL are referred to as program modules, and are programmatically associated with identifiers (handles) to distinguish them within a process.

Executable code required for the simulation model was therefore placed in a DLL, and loaded by the simulation application at the same time as the simulation model's data was loaded. Additionally, the majority of the code directly associated with building and running a simulation model was moved out of the main application to a second DLL called the simulation library. The main application was to provide the user interface and graphics, and other functionality not directly related to computationally simulating an aircraft. The model library provided a convenient location to place functionality used by various tools not directly related to running the simulation. Applications other than the simulation application could load the simulation library and immediately gain all the functionality needed to operate on a simulation model. Figure 1 shows the general structure of the planned simulation environment.

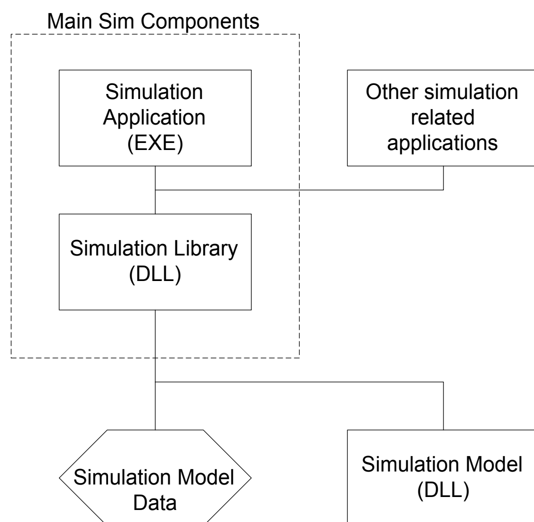


Figure 1. General simulation architecture

Moving the simulation model program code out of the simulation application and into a DLL required a means of sharing data between the two modules. With the performance and resource constraints of an application running on an early Windows based PC, it was not possible to pass data as function parameters. The overhead of loading and unloading the stack had a dramatic effect on performance on the 486 33MHz machines used in the original design implementation.

A common database was used instead, that was shared by all modules within the running simulation. Since simulation application, library, and model DLL all share a common address space within the OS, the problem became more logistical than technical. Data and function pointers are commonly passed between modules in Windows applications, such that it is relatively simple to allow every module within a given address space to access a given variable. However, knowledge of the variable must exist at compile time. The simulation library was given control of a simple heap structure that exported pointers to key elements within the heap that would allow different modules access to the database. For example, a simulation model's DLL could access a variable "Alpha" with the following line of C-code:

```
pSimData [ 0 ] [ GetIndexToName ( "Alpha" ) ] = MyAlpha;
```

Note, the first array index "[0]" is required because Windows can only export pointers to data. pSimData is an array in the simulation library, so it is exported as a pointer to an array. The first pointer resolution is required in the simulation model DLL to convert pSimData back into a simple array.

While this solved the logistics, it had an undesirable impact on simulation performance. Each variable accessed within a module other than the simulation library would be required to indirectly address an array through a string search function. Performance was increased marginally by looking up the required names prior to performing a simulation run, and caching the offsets in a different array. The resulting lines would appear as follows:

```
// Before performing a simulation
iSimData [ ALPHA_INDEX ] = GetIndexToName ( "Alpha" );

// During a simulation
pSimData [ 0 ] [ iSimData[ ALPHA_INDEX ] ] = MyAlpha;
```

On Intel 486 and Pentium processors, the effects of this indexing scheme were somewhat surprising. The processor maintains a single pass indexing capability for up to four levels of indirection. In effect, the processor should be capable of reading the desired value in the above line in a single read operation as long as there are no more than four different offsets associated with the

read operation. However, the line of code actually has 5 levels of indirection: *PSimData, [0], iSimData, [ALPHA_INDEX], and [iSimData [ALPHA_INDEX]].* The resulting machine code two read passes, and causes two separate cache flushes in this single line of code. As stated above, the performance increase over a text-based index search was only marginal.

By assigning the value of *[iSimData[ALPHA_INDEX]]* at compile time, the level of indirection in the previous line could be reduced to three. To implement this, a preprocessor was implemented in the simulation application to generate source code definitions for each shared variable. The above lines could then be converted to the following:

```
// source definitions generated by the simulation application
#define Alpha pSimData [ 0 ] [ ALPHA_INDEX ];

Alpha = MyAlpha;
```

This approach allows the database to be defined completely within the running simulation library by the user and preprocessed to create variable definitions in the simulation model's DLL. The simulation model's DLL is then compiled and loaded into the running simulation application. The entire process requires less than a minute in most cases, and is only required when there is an addition or deletion of a variable in the database that is needed by the simulation model's DLL. The simulation performance of this method approaches that of the traditional combined simulation application and model.

While sharing a common database between the simulation application and the simulation model solves a number of performance related problems, it is a significant deviation from traditional object oriented programming (OOP) techniques in that it provides no encapsulation. The simulation model developer is free to directly access data inside the heap object created within the simulation application. However, the preprocessing capabilities built into the simulation application provide a simple approach to accessing variables in the database that is consistent with OOP methods. Simulation model developers are also discouraged from accessing the database directly. In any case, the performance gains obtained from sharing the database in this manner were considered significant enough to justify this deviation from OOP.

While the running application provides the interface to add and delete new variables, for performance reasons it is restricted to times when the simulation is not running. An interface defined as a heap manager is exported by the simulation application to support database manipulation in a more object-oriented manner than when the simulation is running. In effect, there are functions available for adding and deleting variables, performing searches, and classifying data types. Encapsulation is performed at the interface level rather than in the language. All modules are required to manipulate the database through the provided interface, since the exported database variables only point to the current heap during a simulation run, and performance is not a critical issue while the simulation is not running.

The simulation model was separated into two basic components. The first was the executable functions built into the model's DLL, and the second was the database used by both the application and model code. Additional data sources included smaller blocks of data, such as initial conditions, control and stick deflections, etc. The original design specified a different file, and extension for each data type, such that each was stored and loaded as a separate entity. However, preliminary testing showed engineers found the overhead required to get the simulation up and running to be somewhat oppressive. To simplify simulation operation, all of the files were stored in or referenced by a single *project* file. The database, initial conditions, and table look-up information were placed directly in the project file, while the model's DLL and the tabular table data were referenced by file name to reduce the size of the project file. Support for each of the separate file types was also retained to allow models to be assembled from different pieces, so different individuals could work on different portions of a model simultaneously. Figure 2 shows the different components of a typical simulation model.

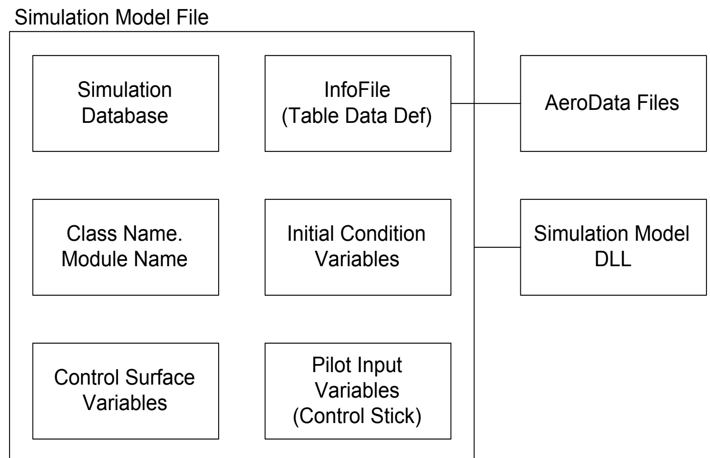


Figure 2. Components of a simulation model

Aerodynamic Database

Probably the most difficult problem encountered was developing a data table structure that supported both

rapid prototyping and full simulation model development, and did not prevent the simulation from operating in real-time. Preliminary design goals required support for tables of any dimension that could be resized and edited without source code changes. To support in-house editing tools, and to improve source control operations, the actual data was to be stored as a number of two-dimensional data files that included indexes for both dimensions and an index value for an optional third dimension.

BAR had already adopted a two-dimensional data file format as the basic building block for all data tables of any dimension. A number of in-house tools had been developed to manipulate the data in the files, and provided useful functionality that was not planned for the first version of the simulation application. The existing format was therefore included with only minor changes as the supported internal data table format (BAR 2.0). The design included a second definition file (InfoFile) to define multiple dimensional tables as groups of two-dimensional data files. The InfoFile contained the names and order of the data files used in each look-up operation, as well as table dimensions, independent variables and other information not contained in the BAR 2.0 files. Since assembling the table data was considered one of the more complex tasks of developing a simulation model, a significant effort was expended to produce a simplified visual interface for the task. Figure 3 shows the basic structure of a typical look-up table in the InfoFile.

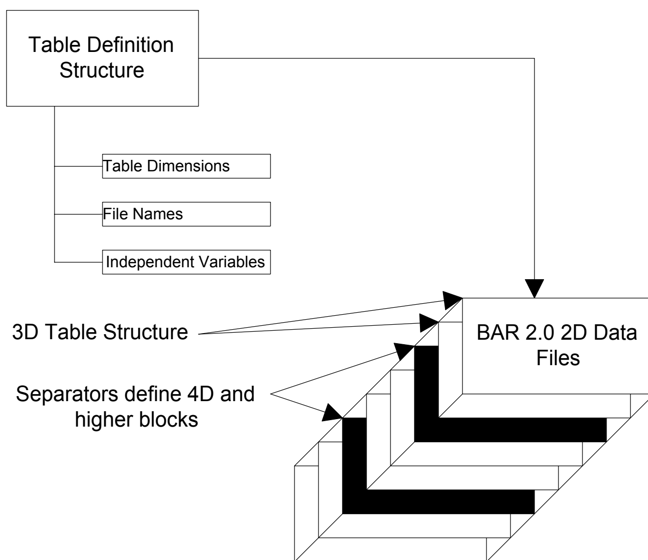


Figure 3. InfoFile Table Definition

Since the InfoFile was designed to support the complex editing features of a visual interface, table look-up operations were predictably too slow to support real-time simulation. A second data table format (DTF) was developed in parallel to the InfoFile to provide the actual table look-up functionality during a simulation. A DTF file is essentially a compiled form of the InfoFile and related BAR 2.0 data files. A number of performance optimizations were implemented, such as redundant index and table removal, last value comparisons, and index boundary crossing detection. Since the DTF file data could always be regenerated from the InfoFile, it could be optimized without regard to user access, data editing, or table identification. A highly optimized table look-up engine was developed to work with the DTF file data that required an average of 75 nanoseconds to perform a table look-up on a single dimension of a typical data file. Some additional optimization was achieved by restricting tables to a maximum of sixteen dimensions, and a small section of the table look-up code was converted to assembly language when the C++ compiler failed to produce the expected optimizations.

Simulation Model Extension Interface

Flight simulation technology is still an evolving technology. A significant amount of research is still being performed to better model different aircraft, and new techniques are emerging constantly. Because of this, the simulation application was designed to allow simulation models to replace any simulation functionality normally provided by the application. In effect, the simulation model could replace any function contained within the simulation application that was called during a simulation. This was to allow simulation technology researchers to test their theories using a stable, well-tested simulation application, and to compare their results with existing methods. The researchers could then concentrate on only the portions of the simulation their research involved without developing a new application, and with insurance the rest of the application was running as expected.

To support the replacement of simulation functionality within a simulation model, the simulation application would search the list of exported functions within the simulation model's DLL for a name matching a specific set of replacable functions. If a replacable function was found, the application called the function provided by the simulation model rather than its own. The researcher had only to provide a function with the correct name, and the source code for the replacable functions were made readily available as a starting point to further reduce the learning curve. A simple list of function pointers was used to reference the correct functions. While there was a significant increase in function call overhead compared to calling a function directly, the number of required indexed function calls was too small to measurably affect simulation performance.

External Development and Control Interfaces

The addition of extensibility to the simulation application was not restricted to simulation technology research. A real-time high fidelity simulation has a number of applications in other fields, such as training and interactive war gaming, etc. The simulation design included several interface specifications to load specialized or generic modules to implement otherwise unsupported functionality. An I/O interface was included to provide a relatively simple method of connecting the simulation to various instruments, switches, and control devices. A graphics interface was included to support external displays, intercoms, and sound devices. Finally, a generic interface was included to support unknown functionality. The graphics and I/O interfaces are derivations of the generic interface. The primary difference being optimizations built-in to the simulation application to support the non-generic implementations with less overhead.

The generic interface supports starting and stopping the simulation, loading and saving simulation models, and menu additions. It was originally intended as a supporting interface for networking and control software. However, several tools have been developed and incorporated into the simulation application using the generic interface. For example, a control system design tool capable of reading block diagram specifications and converting them to source code was implemented in this manner. Another tool is currently being developed to automate the process of editing and compiling a simulation model's source code.

Networking modules have also been implemented using the generic interface. However, an extension to the interface, the Module Communication Interface (MCI), was required to manage data definitions, conversions, and message routing. MCI provides network aware modules with registration and message services, and maintains a separate, reentrant database interface for storing time-dependent data values for different objects participating in a simulation. MCI does not provide network services, define networking protocols, or require specific data types or formats. Instead, it provides a common interface suitable for binding the communication and data definitions used by most networking services. For example, an existing proprietary network interface developed specifically for verifying the functionality of the generic interface and MCI is not directly compatible with the Distributive Interactive Simulation (DIS) standard. However, a DIS compatible MCI module can communicate with objects communicating over the proprietary networking interface through MCI. In effect, MCI is a transport layer designed to bind different network protocols together, and is optimized for use with interactive simulation applications.

Component Object Models

All of the interfaces designed for the simulation application were implemented as a list of functions exported by the module implementing the interface, and imported by the modules connecting to the interface. While this approach offers encapsulation, with the previously discussed exceptions, it lacks true polymorphism. The behavior of an interface function can be changed by replacing the DLL that implements it, but that would mean altering known behavior of the interface. A more suitable OOP approach for defining behavior between modules is with a component object model (COM) specification.

While Microsoft has supported COM through a proprietary interface specification called object linking and embedding (OLE) since the release of Windows 3.1, BAR has not adopted it due to the number of alternative specifications used on different (non-PC) platforms, the incompatibilities between them, and the fear the Microsoft would discontinue or significantly alter OLE/COM. However, recently several UNIX operating systems on PC's and on workstation computers have agreed to support the OLE/COM interface, and Microsoft plans to convert the traditional Windows GDI into a COM interface in the next release of Windows NT.

BAR plans to provide COM interfaces as alternatives to all existing interfaces for version 2.0 of the simulation application (D-SIX) scheduled for release in January 1998. This will provide a means of distributing the functionality of a simulation across multiple computers without a proprietary interface specification. Additionally, the interface will be platform independent, allowing transparent interoperability with other computer hardware not running Windows. For example, the program code for a simulation model could be implemented on a different computer with a different operating system than the computer running the simulation. The interface specifications will be made freely available to any interested parties.

Performance

Throughout the development of the simulation software, the initial design has remained surprisingly constant. The restrictions imposed by the limitations of the PC/Windows environment required a number of compromises internally to address performance problems. However, the interfaces between each program module remain almost identical to those originally designed. The

most significant impact of the PC/Windows environment was the amount of time required to optimize the source code, since the simulation design was originally intended to operate in real time on 486 machines.

With the performance of PC's today surpassing the performance of workstations used for simulations less than three years ago, source code optimization has become more of a habit than a requirement. However, a high fidelity simulation has applications in training and interactive simulations that require additional processor time. The ability of the simulation to function in real-time while running add-on modules for high-resolution graphics and interactive networking can be attributed primarily to the rigorous optimizations implemented throughout the development process.

The current implementation of the simulation application (D-SIX) is capable of running extremely complex simulation models in real-time on most standard PC's. For example, the F18E/F model imported into D-SIX from the MDA model contains more than eight hundred multidimensional look-up tables and almost four thousand database variables (1.4 million data points). The simulation runs the F18E/F model in real-time at 80Hz on a 166MHz Pentium machine. This includes a port of the full flight control system consisting of more than thirty thousand lines of Fortran code.

User Interface

One of the advantages of developing software on an inexpensive PC is the relatively low cost of advanced development tools. The large number of software developers producing software for the PC has allowed compiler manufacturers to expend enormous resources on producing advanced development tools and still distribute them at relatively low prices. Additionally, since the majority of the applications for the PC are for non-technical fields, such as business applications and games, development tool advancement has been largely in the direction of improving user interfaces and simplifying usage through graphical or *visual* interfaces. The development of D-SIX profited significantly from these tools. Simulation model development and analysis complexity was reduced considerably compared to more traditional simulation applications.

The display interfaces shown in figure 4 illustrates a few of the graphical user interfaces used in the simulation environment. These interfaces control operations ranging from simulation independent variable definition and input, model initial definition and setup, control surface time-history editing, as well as graphical editing of any table-based terms. It is the extensive use of graphical interfaces such as these that allows the engineer to conduct virtually all of the typical simulation development, analysis, validation, and deployment operations with little or no code level interaction.

Conclusion

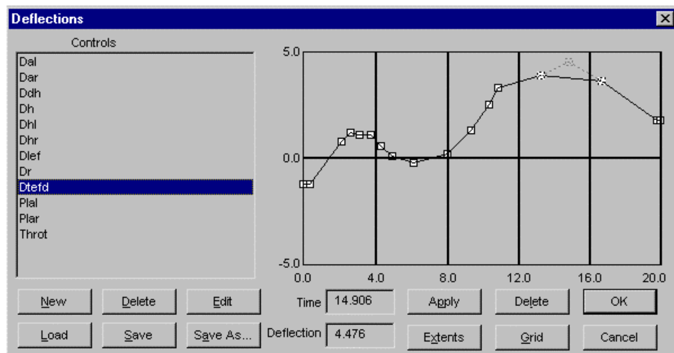
Performance issues related to PC limitations were a significant consideration on the original software design. While current PC performance has increased by orders of magnitude from the original design platform, coding efficiencies have yielded the current capability to host the most complex engineering simulations in real-time on a PC.

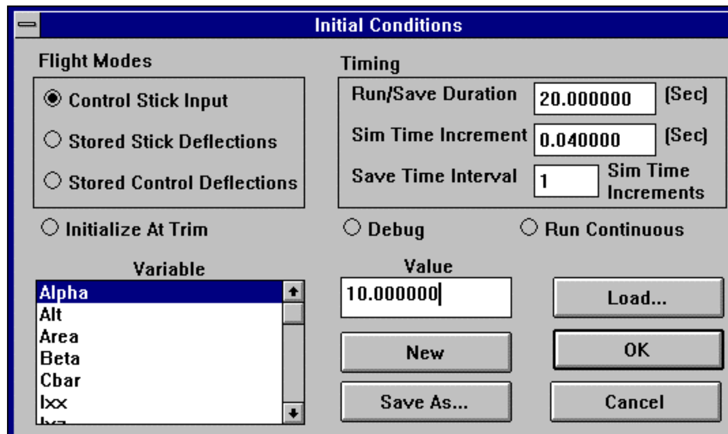
The idea of the original design was to produce a simulation platform that could be easily expanded as hardware and operating system technology advanced. While new technology and better PC performance has significantly enhanced the capabilities of the software, the original interface specification has changed very little. The flexible interface design has allowed the rapid development of new capabilities, such as high-resolution graphics, and interactive networking. New capabilities and new simulation technology can be added without changes to existing software through add-on modules and simulation model extentions. As long as developers produce software compatible with the existing interface design, compatibility of simulation models, add-on modules, and the simulation application is guaranteed.

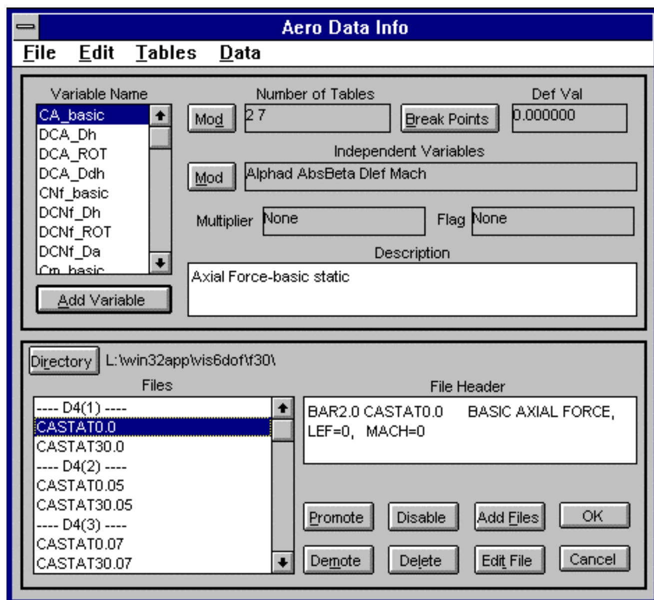
References

Petzold, C., *Programming Windows: The Microsoft Guide to Writing Applications,* Microsoft Press, Redmond, WA, 1992. ISBN 1-55615-395-3.

Brockschmidt, K., *Inside OLE,* Microsoft Press, Redmond, WA, 1995. ISBN 1-55615-843-2.







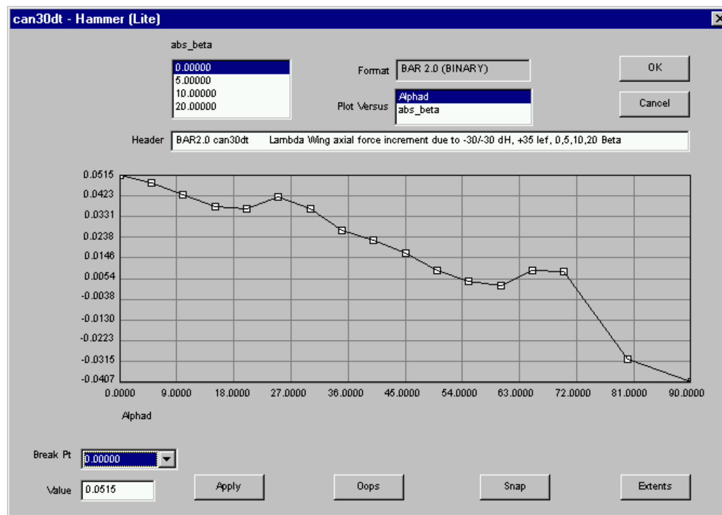


Figure 4. Examples of graphical editing features in D-SIX