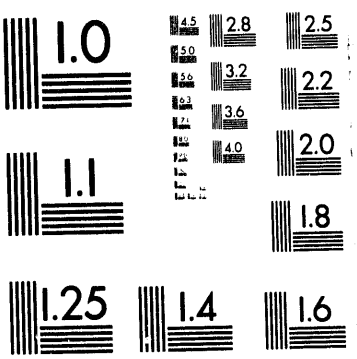
1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0

1 of 1

UCRL-LR-116004
Distribution Category UC-705

On the Implementation of Error Handling in Dynamic Interfaces to Scientific Codes

Cynthia Jean Solomon

(M.S. Thesis)

Manuscript date: December 1993

LAWRENCE LIVERMORE NATIONAL LABORATORY
University of California • Livermore, California • 94551

On the Implementation of Error Handling in Dynamic Interfaces to Scientific Codes

By

Cynthia Jean Solomon
AA, Southwestern Oregon Community College, 1988
BS, Western Oregon State College, 1991

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Nelson S. Max

Committee in Charge

1993

Cynthia Jean Solomon
December 1993
Computer Science

On the Implementation of
Error Handling in Dynamic Interfaces to Scientific Codes

Abstract

With the advent of powerful workstations with windowing systems, the scientific community has become interested in user friendly interfaces as a means of promoting the distribution of scientific codes to colleagues. Distributing scientific codes to a wider audience can, however, be problematic because scientists, who are familiar with the problem being addressed but not aware of necessary operational details, are encouraged to use the codes. A more friendly environment that not only guides user inputs, but also helps catch errors is needed. This thesis presents a dynamic graphical user interface (GUI) creation system with user controlled support for error detection and handling. The system checks a series of constraints defining a valid input set whenever the state of the system changes and notifies the user when an error has occurred. A naive checking scheme was implemented that checks every constraint every time the system changes. However, this method examines many constraints whose values have not changed. Therefore, a minimum evaluation scheme that only checks those constraints that may have been violated was implemented. This system was implemented in a prototype and user testing was used to determine if it was a success. Users examined both the GUI creation system and the end-user environment. The users found both to be easy to use and efficient enough for practical use. Moreover, they concluded that the system would promote distribution.

Keywords: GUI, GUI creation system, constraint system, minimum evaluation scheme, naive checking scheme.

Acknowledgments

Among the many people who supported me along this extensive journey, I would like to specifically thank the following:

My Husband: Bob, thank you for enduring the separations, the anxiety, and the stress and for festooning me with your love and support (not to mention your outstanding editing skills). You are a special man. I love you.

My Children: Adam and Nora, a special thanks to you two. Having no choice in the situation you were placed, your kindness, support, and love demonstrate an unselfishness beyond your years. My pride and love for you are immeasurable.

My Parents: I don't know how you did it, but somehow you managed to raise me believing that I could achieve whatever I wanted to achieve. Thank you.

My Advisors: Thanks for everything. Scott, I can't tell you how much I appreciate everything you have taught me and everything you have done for me. I feel extremely fortunate for having had the opportunity to work with you. Pat, thank you for the enthusiasm you had for my project and thesis. It was there every time I needed it.

The DOE: This work was supported by the U.S. Department of Energy, by Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Historically, computational scientists have focused on the design and implementation of state-of-the-art algorithms and methods while largely neglecting the user environment. That is, many scientists have been interested in the speed and the results of their applications but less interested in how intuitive their applications are to use. This has resulted in "user-hostile" scientific codes, easily usable only by those who develop them. Recently, with the advent of powerful workstations with windowing systems, the scientific community has become more interested in user friendly interfaces as a means of promoting the use of their scientific codes by colleagues.

Distributing scientific codes to a wider audience can, however, be problematic because scientists, who are familiar with the problem being addressed but not aware of necessary operational details, are encouraged to use the codes. For instance, a physicist attempting to use a code that models a tokamak fusion reactor is probably an expert in magnetic fusion. All tokamak experts know that a tokamak has a "major radius" and "minor radius" and what they are. However, it is possible they might not know that the variables within the code representing the major radius and minor radius are respectively `rmajor` and `rminor`. This example illustrates a gap that may exist between developers and users. This gap may make using the code much more difficult, if not impossible, and thus, impede distribution.

Bridging the gap between developers and users requires a more friendly user environment. Graphical user interfaces (GUI's) can help provide this by furnishing many helpful features. For example, an input window with entry fields that guide input relieves the user from the necessity of knowing variable names. This GUI feature solves the unknown variable name problem that confronted the physicist in the tokamak fusion reactor

code example. Other helpful features include dialog boxes for prompting users for responses or input, popup boxes for presenting informational messages, check boxes for making exclusive or non-exclusive selections, buttons and menus for easy action execution, and help systems for additional guidance.

Although GUI's help create a friendlier user environment, a truly friendly code also helps catch errors. Recall the physicist attempting to use the code that models a tokamak fusion reactor. It is possible the physicist does not know that a model the code supports becomes less accurate if the major radius is not twice the minor radius. Therefore, the user may unknowingly commit an error that produces inaccurate results. Since placing a GUI between the user and scientific codes promotes casual use, error detection and handling is essential.

We are interested in scientific codes with certain characteristics:

- *Complex.* Many (100's) of inputs and outputs. Code can solve a variety of different problems.
- *Interactive.* Users direct code calculations and query code for information in real-time. As a result, there is no clear flow of control.
- *Programmable.* Users can add to and modify the code at run-time.

Examples of this kind of scientific codes are SUPERCODE, a code for modeling and optimizing designs of tokamak fusion reactors [1] and codes written using the Basis System [2] such as CORSICA [3], a comprehensive tokamak simulation code.

Scientists use these codes as tools to define a model, simulate it, and analyze the results. Because these codes can have hundreds of inputs, one static interface presenting entry fields and menus for all possible settings would be too large to be usable. Furthermore, when scientists use these codes, they usually focus on a particular problem of interest dealing with a smaller subset of the inputs. Providing GUI's for every possible subset problem a scientist may wish to examine is, if not impossible, too time consuming and expensive. Current GUI creation systems have been quite successful for many scientific applications, but they produce static GUI's and do not directly support the necessary error detection and handling. A solution is a GUI creation system with user controlled support for error detection and handling. Such a system aids the distribution of scientific codes by providing a user environment that is both robust and easy to use.

1.1 The Prototype

We have created a prototype system allowing dynamic GUI creation and error handling for initial use in SUPERCODE. Figure 1 displays a schematic representation of the SUPERCODE architecture. Notice that SUPERCODE is a distributed application consisting of a front-end and a computational kernel. The front-end and kernel communicate via a high-speed network link. The front-end consists of a dynamic GUI with error handling and graphics facilities, while the kernel consists of physics and engineering modules coupled with a powerful, programmable shell. This shell understands a subset of C++.

The front-end is built using the OI tool kit, a GUI class library from Solbourne of ParcPlace Boulder [4], and ag/X Toolmaster, a graphics library from UNIRAS [5]. The GUI front-end is equipped with a terminal window that enables direct access to the shell. The GUI creation system is implemented as part of the GUI front-end and the shell. This system enables the front-end to dynamically display additional interfaces in response to messages from the shell.

The shell controls all aspects of code operation. The shell operates in an interactive mode sending results to the GUI terminal window in response to user inputs. The user enters some text, the GUI sends the text to the shell, the shell processes the text executing any physics and engineering modules necessary, the shell sends any output to the GUI, and the GUI displays it in the terminal window. Within this flow of control, the shell and the GUI must work together to provide GUI error handling.

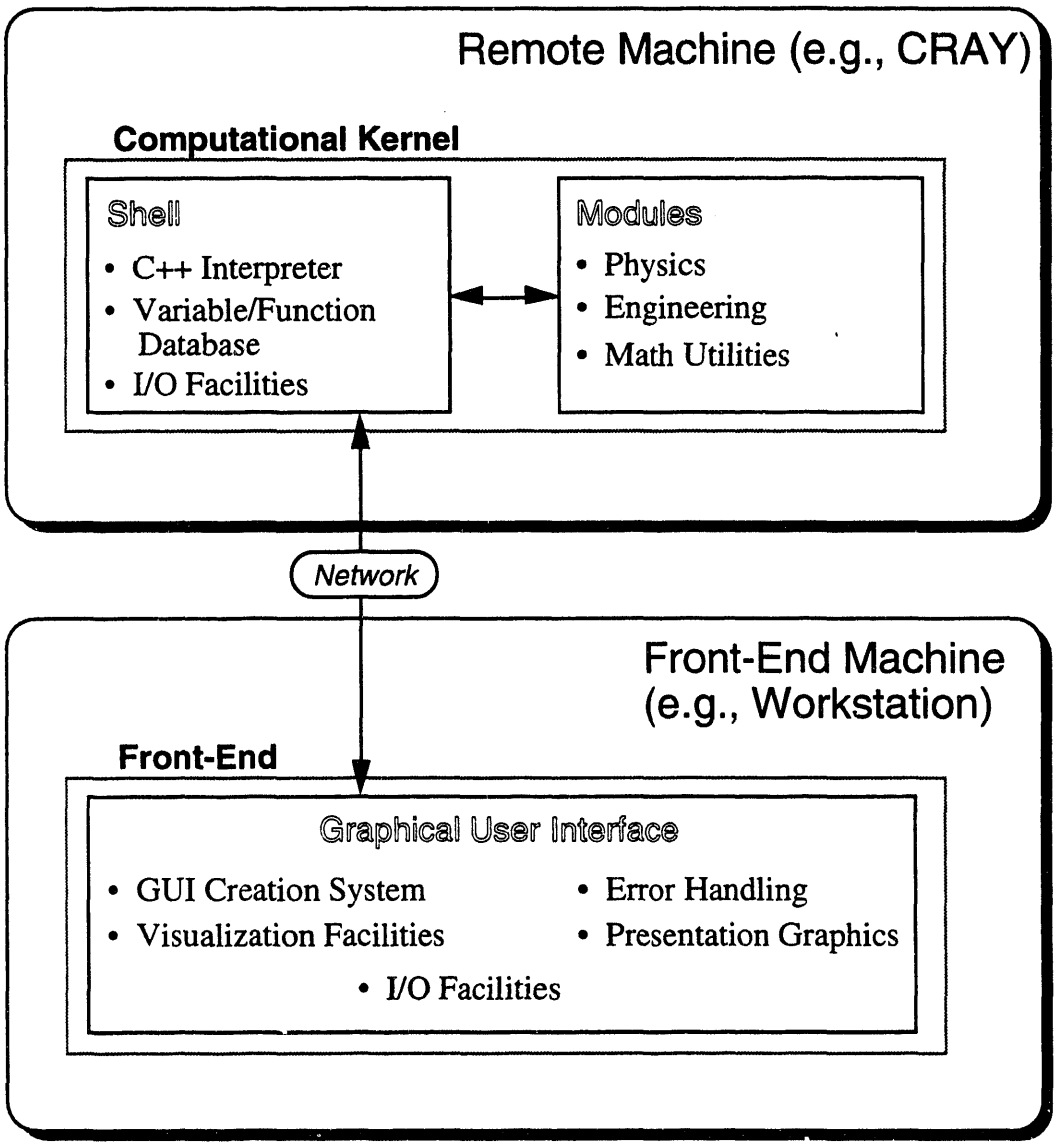

Figure 1: SUPERCODE *architecture: modules, shell, graphical interface [1].*

1.2 Approach

The GUI creation system and its error handling mechanism are integrated into the SUPERCODE GUI front-end and shell to provide GUI's with error detection and handling support. The error handling mechanism is called a *constraint system* because error detection is accomplished by defining and examining constraints. A constraint has two parts: a condition clause for error detection and an action routine for error handling. An expert GUI designer/modeler creates an interface and defines appropriate constraints to

support the particular physical model the interface represents. The shell stores the error condition and handler information and sends the GUI the necessary information to support the system. The shell detects error conditions in the user's input to the GUI and uses the handler to respond with an appropriate action such as notifying the user.

A key requirement of the GUI error handling system is that users be notified as soon as possible when they commit an error. To support this, the error conditions defined by the constraints are checked every time the execution state changes (i.e., the user has entered a new value for a variable). Since there can be hundreds of constraints defined and few, if any, related to a single change, checking every constraint every time can result in a considerable time cost. Long delays affect the usability of the system. We call checking every constraint every time a *naive* checking scheme. Since speed is an important usability factor, a *minimum evaluation* scheme has been examined and implemented. We will show that this minimum evaluation scheme significantly reduces the time the GUI spends trying to detect errors.

1.3 Evaluation Metrics

Perhaps the most important metric is whether the constraint system aids users. The constraint system is considered a success if the following statements hold true:

(1) Creating a GUI for a code and defining constraints that enforce its model are easily accomplished by the GUI designer.
(2) The casual user can use dynamically displayed GUI's to use SUPERCODE safely without knowing the specifics of the complete model.
(3) The constraint system is fast enough for efficient use.

Point (1) is an important issue. Before a casual user can use a dynamically configured GUI, an expert GUI designer must create it. If this process is too difficult, the designers won't bother. Point (2) addresses whether or not the error handling mechanism works. If casual users unknowingly commit errors and produce invalid results, the system is useless. Point (3) addresses another important metric: run-time performance. If the system is too slow, no one will use it. What is the run-time performance of the naive checking scheme? What is the run-time performance of the minimum evaluation scheme? Which is better, and is either of them fast enough to satisfy Point (3)?

1.4 Organization

The thesis is presented in four parts: Chapter 2 addresses related work in the areas of GUI creation systems and exception handling and shows why this previous work, although related, is inappropriate for this environment. Chapter 3 shows how we implemented a GUI definition language to provide a GUI creation system and how we extended it to provide error handling by implementing a constraint system. It further describes the language extensions needed to simplify the constraint system implementation. Finally, the chapter details a minimum evaluation scheme. This optimization reduces the run-time that the constraint system requires to detect errors. Chapter 4 discusses the success of the GUI system -- specifically how easy it is to use. The success of the minimum evaluation scheme is also evaluated. Execution times are presented for checking the constraints of several GUI's for both a naive detection scheme and the minimum evaluation scheme. A comparison between the execution times of the two schemes demonstrates the utility of the minimum evaluation scheme. Finally, the success of the end-user environment is evaluated by presenting user feedback. Chapter 5 offers suggestions for enhancements and directions for future work.

Chapter 2

Related Works

Current GUI creation systems can provide some, but not all, of the facilities required to build a robust front-end to a scientific code. Error detection and handling mechanisms are among these facilities. Error detection and handling mechanisms do exist and have been implemented at the programming language level. However, these implementations do not support the kind of error detection and handling needed for the scientific code environment. We examine current GUI creation systems along with current error detection and handling mechanisms and describe why these existing mechanisms do not suffice.

2.1 Current GUI Creation Systems

Examples of current GUI creation systems include DevGuide [6], the NeXTstep Interface Builder [7], Garnet [8], and AVS [9]. DevGuide and NeXTstep Interface Builder are User Interface Management Systems (UIMS's). UIMS's are fairly easy to use and allow the quick development of static interfaces designed for particular applications by providing graphical tools to aid the GUI development process. A designer creates GUI components by selecting icons. For example, to create a window, the designer selects the icon that represents a window. To put a button in the window, the designer first creates the button then drags it into the window. Facilities exist for setting additional component attributes: background color, size, border width, etc. Once the GUI designer has the desired interface on the screen, it can be saved to a file, processed, or both. If it is saved to a file, the designer can retrieve it for additions or modifications. When the GUI is processed, the

UIMS's produce code that implements the desired interface. However, before the generated code can be compiled and used with an application, it must be modified. For instance, UIMS's do not know the names of the functions defined within the application for which the GUI was created. If a button is created to execute one of these functions, appropriate modifications must be made to the generated code. The code that implements the underlying application must be added as well. These additions and modifications are called "hooks." UIMS's only generate the code that displays the interface while a programmer must provide the hooks into the underlying application. After the designer has added the necessary code to the UIMS generated code, it is compiled, and a GUI that is specifically tailored for that particular application is created.

Garnet is also a UIMS, but it has an additional feature. The GUI's created by Garnet can be manipulated by the user (i.e., a user can change a button's position). The resulting interface, however, is still static. Although these UIMS's are valuable tools, they are inadequate for our needs because they are not dynamically configurable, nor do they provide error detection and handling mechanisms.

AVS is not a UIMS. It is a framework application that can be used to develop interactive scientific visualization applications. Flow networks of existing software building blocks or modules are created by connecting them using a direct-manipulation user interface. AVS can generate a simple user interface to each module as well as a simple user interface for the flow network. Unfortunately, AVS is too limited for our needs. AVS modules are restricted in the types of data they can support (i.e. graphical data such as vectors), and these modules are limited to an insufficient six inputs. Also, the interfaces AVS creates are too simple for our application, and error detection and handling mechanisms are not supported.

2.2 Language Level Error Handling

Although user controlled error detection and handling is not supported by current GUI systems, many programming languages systems do support it. Goodenough [10] provided the theoretical foundations for language level exception and error handling. Many of these techniques have been used to implement language level exception handling. PL/I [11], CLU [12], ADA [13], and Mesa [14] are examples of procedural languages with exception handling features. Exception handling models have been proposed for the object-oriented paradigm by Yemini [15] and Dony [16]. Some examples of object-oriented languages

with exception handling are C++ [17], SmallTalk [18], Eiffel [19], and Lore [20]. An exception handling model emphasizing parallelism was proposed by Levin [21]. We examine Goodenough's theoretical foundations for language level exception and error handling and describe how we apply these foundations to the prototype implementation.

2.2.1 Relative Language Level Error Handling Concepts

Goodenough provided a methodical theoretical analysis of exception handling in programming languages. He identified and defined the key components of exception handling. He also detailed reasons why programmer controlled systems should support exception handling and the requirements and issues the systems should address. Goodenough defined exception conditions as those conditions detected while a function is attempting to perform some operation which are brought the attention of the function's invoker. These conditions may or may not require some action by the operation's invoker and therefore, must be brought to the invoker's attention. He called this *raising an exception*. Once the exception is raised, the invoker responds in some manner (including taking no action). This response is called *handling the exception*.

Goodenough defines three main reasons why a system should support programmer controlled exception handling (the labels are this author's): [10]

> [RESPONSE] to permit dealing with an operation's impending or actual failure.
>
> [INTERPRET] to indicate the significance of a valid result or the circumstances under which it was obtained. In this case, the operation's result satisfies its output assertion, but the invoker needs additional information describing the result before he can give it an appropriate interpretation. For example, addition overflow on many computers produces a valid result as long as the bits of the result are interpreted appropriately.
>
> [MONITOR] to permit the invoker to monitor an operation, (e.g., to measure computational progress of a computation or to provide additional information and guidance should certain conditions arise).

Goodenough also lists the four requirements and issues a programmer controlled exception handling should address (the labels are this author's): [10]

[ASSOCIATION] *Association of handlers with invocations of operations.* Since exceptions occur when attempting to perform some operation, one basic issue is how to associate the proper handler with the invocation of a given operation.

[CONTROL FLOW] *Control flow issues.* These issues concern how to ensure that the user and the implementer of an operation agree on whether termination or resumption of an operation is permitted when a particular exception is raised, and how a programmer expresses which of these possibilities is being chosen.

[DEFAULTS] *Default exception handling.* It is useful to provide default handlers for exceptions raised by an operation but not handled by an invoker of the operation.

[HIERARCHIES] *Hierarchies of operations and their exceptions.* Exception handling issues that arise from the interaction between an exception raising operation and its immediate invoker are somewhat different from those that arise when an exception is disposed of by an indirect invoker.

2.2.2 Exception and Error Handling in the Prototype

Exception and error handling in the prototype parallels Goodenough's definitions and theories. The difference is that Goodenough's model targets programming language systems whereas the prototype is a user-controlled system.

In the prototype, exception conditions are defined for a GUI by a GUI designer. Exception conditions are detected when the system performs a consistency check. Just as in Goodenough's model, the prototype exception (error) may, or may not, require some action on the part of the user and should be brought to the user's attention. However, there is one area in which the prototype system differs from the Goodenough model. Goodenough insists that the invoker, which is analogous to the prototype's user, should be notified. This requirement is not enforced by the prototype because the GUI designer provides the specific actions that occur when an exception condition is detected.

The GUI designer should, but does not have to, provide the user with an opportunity to respond to the detected exception condition. We first examine the model that notifies the user. An exception condition is detected, and the GUI designer provides actions that notify the user by displaying a message. Displaying a message is analogous to raising an exception in Goodenough's model. The user's response to the violation message is handling the exception. The user has the option of either responding (i.e.,

entering a different value in an entry field) or not responding (ignoring the error). The prototype differs from Goodenough's model when the system does not notify the user (invoker) of a detected exception condition. Suppose that the GUI designer provides an entry field for the variable `X` and defines a constraint on `X` to be greater than `0`. Suppose also that the GUI designer defines exception handling actions that do not include notifying the user. Instead, when the constraint is violated, `X` is set to `1`, and the program continues. Setting `X` to `1` is handling the exception, without the invoker ever being notified. Another possible scenario is that the GUI designer provides action statements that execute additional statements and also notify the user. We consider it bad technique if the GUI designer does not notify the user, and discussions throughout the rest of this paper make the assumption that the user is notified. This is how we applied the definitions presented by Goodenough to the prototype system.

One of the reasons Goodenough presented in support of programmer controlled exception handling applies to the prototype. The type of exceptions (or violated constraints) that occur in the prototype is related to [RESPONSE]. The user is notified of a condition that could render the results invalid and is permitted to deal with the situation. For example, suppose the system defines an extended family in which the members are given age values. Suppose further that `grandpa` and `mother` are members of this family, and `grandpa` is `mother`'s father. Given this relationship, a constraint is defined requiring `grandpa` to be older than `mother`. If `grandpa` has an age value of `59`, and the user enters an age value of `60` for `mother`, the constraint is violated. In response, an appropriate violation message is displayed directing the user to change one of the values for `mother` or `grandpa`.

[INTERPRET] and [MONITOR] do not present themselves in the prototype system because of the nature of the user's data entry. [INTERPRET] involves raising an exception for a valid result that, for some reason, needs additional interpretation. In the prototype, exceptions are only raised when an invalid entry is made. If a valid entry is made, an exception is not raised because no additional interpretation is necessary. [MONITOR] exception conditions keep track of a computation's progress possibly to provide a method of supplying additional information at certain points. The prototype does not need monitoring because it does not require additional information during the execution of its operations.

Assuming that `mother` can not be older than `grandpa`, given their relationship, the user should respond by changing one of the values. However, there are cases where the user inputs may, or may not, result in an invalid system. For instance, suppose `grandpa` has age value 59, and the user enters the age value 44 for `mother`. This makes

`grandpa` only 15 years older than `mother`. Although this is suspicious, it is not impossible. The designer may wish to notify the user of these situations as well. We consider constraints that define possible errors as *warnings*. Warning constraints define possible error conditions while error constraints define definite error conditions. Also note that the proposed exceptions may be forcibly ignored. Suppose that `grandma` remarried a younger man. Then, `grandpa` may be younger than `mother`. The GUI design prototype handles all three of these situations.

Finally, we examine how the four requirements and issues [ASSOCIATION], [CONTROL FLOW], [DEFAULTS], and [HIERARCHIES] of programmer controlled exception handling apply to the user-controlled prototype. [ASSOCIATION] deals with how to associate the proper handler with the invocation of a given operation. In the prototype, the GUI designer always pre defines the handler associating it with the appropriate operation. Given this simplification, issues [CONTROL FLOW], [DEFAULTS], and [HIERARCHIES] are also determined by the GUI designer.

Chapter 3

Design and Implementation

The GUI creation system enables expert GUI designers to create a GUI and define constraints to impose appropriate restrictions on user inputs. As the system evolves (e.g., an end-user enters inputs that define a system), the constraints defined by the GUI designer help prevent the end-user from defining an invalid system. There were two choices for defining and creating the constraints: modifying the shell to add to the shell language or creating a C++ class. A language extension implemented by modifying the shell would allow the flexibility of choosing a favorable syntax but would not allow the constraint system to be used in compiled code. Creating a C++ class would be easier to implement, but it would require using the longer and more awkward C++ syntax. As a compromise, we chose the C++ class implementation but gave it the flavor of a language extension by using a preprocessor.

There were also two obvious methods for checking the constraints: a naive scheme that checks every defined constraint each time the system changes and a minimum evaluation scheme that only checks those constraints that may have been violated as the system evolves. The first method is naive in that it examines many constraints whose values have not changed each time a user enters an input. The minimum evaluation method requires some overhead to determine which constraints must be examined but often executes fewer constraints than the naive method. In this chapter we examine the design and implementation of the GUI creation system. This system includes the GUI component creation system, the constraint language syntax, the preprocessor, and both constraint checking schemes.

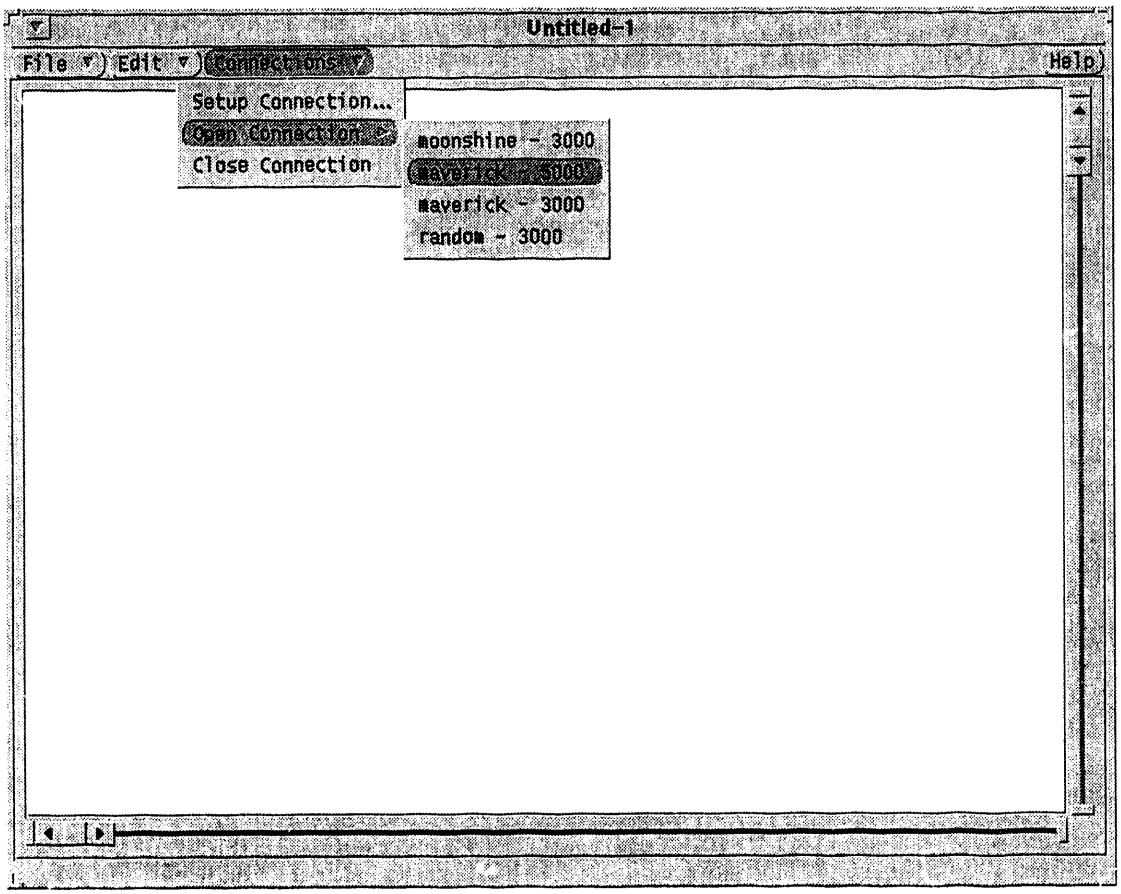

Figure 2: Terminal window with connection menus displayed.

3.1 General Overview

Before examining the detailed implementation of the GUI creation system, it is helpful to understand the overall process. First, an end-user starts the distributed system. The GUI front-end displays a terminal window that enables direct access to the SUPERCODE shell. Using the main menu of the terminal window, the user establishes a connection with this shell. Figure 2 shows the terminal window and its connection menus.

Once a connection is established, a prompt appears in the terminal window signifying the system is waiting for user input. Next, the user asks to display the desired GUI. To display a dynamically configurable interface, the shell must read and process a GUI specification file. Expert GUI designers create the GUI specification files that contain

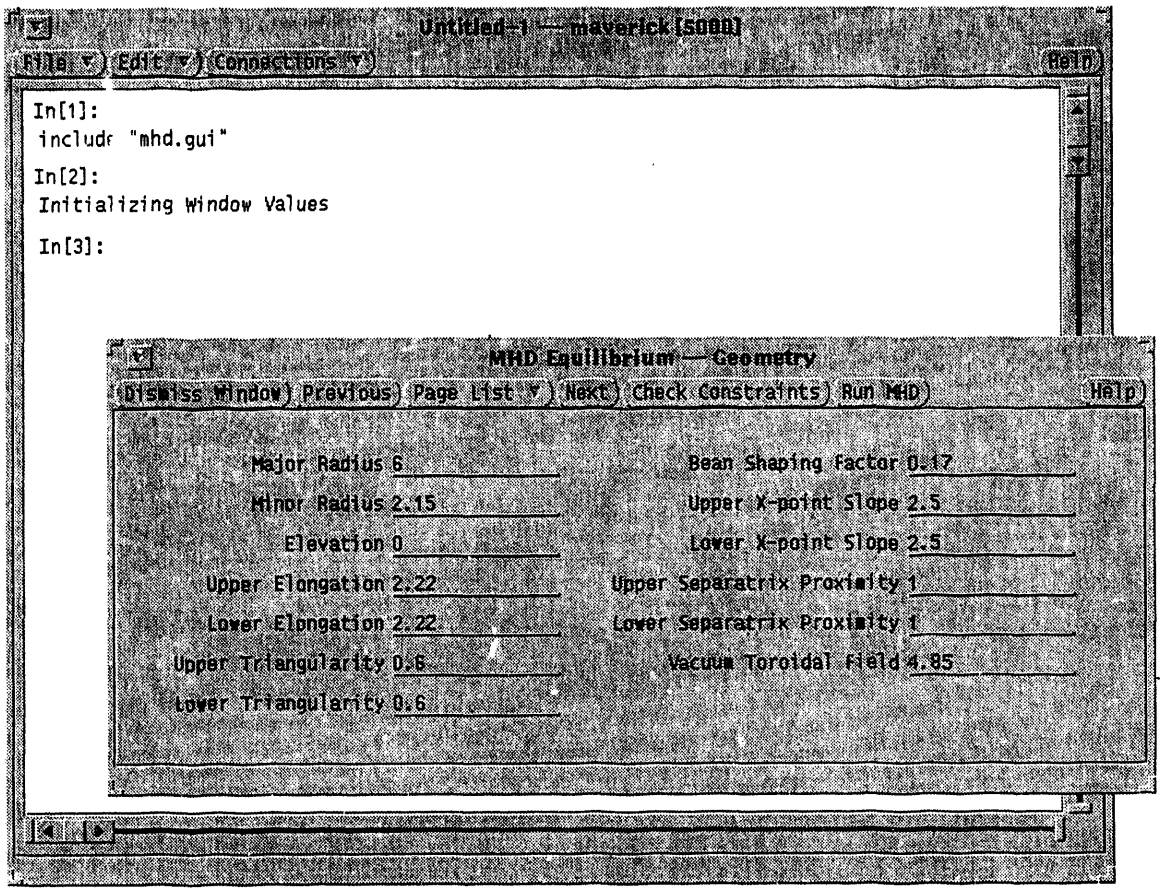

Figure 3: Terminal window and GUI defined in file `mhd.gui`.[1]

the commands which create the GUI's and define the constraints that support error detection and handling. The user uses an `include` command to direct the shell to read the desired GUI specification file. The shell reads and processes the commands contained in GUI the specification code, creating the defined interface and sending the appropriate instructions to the front-end to display it as well as creating any defined constraints to support error detection and handling. Figure 3 shows the terminal window and the resulting interface defined in file `mhd.gui`. When the GUI is displayed, it is initialized by querying the shell for and displaying current system values.

Whenever a user enters a new value into an entry field, the GUI sends this value to the shell with its corresponding variable in the form of an assignment statement. This assignment statement is also displayed in the terminal window as seen in Figure 4. The shell processes the assignment statement which changes the state of the system and

[1] Since the users did not like two adjacent prompts displayed without any indication of what occurred between them, nor did they like displaying the many commands used to initialize the window in the terminal window, the front-end displays the descriptive statement, `Initializing Window Values`.

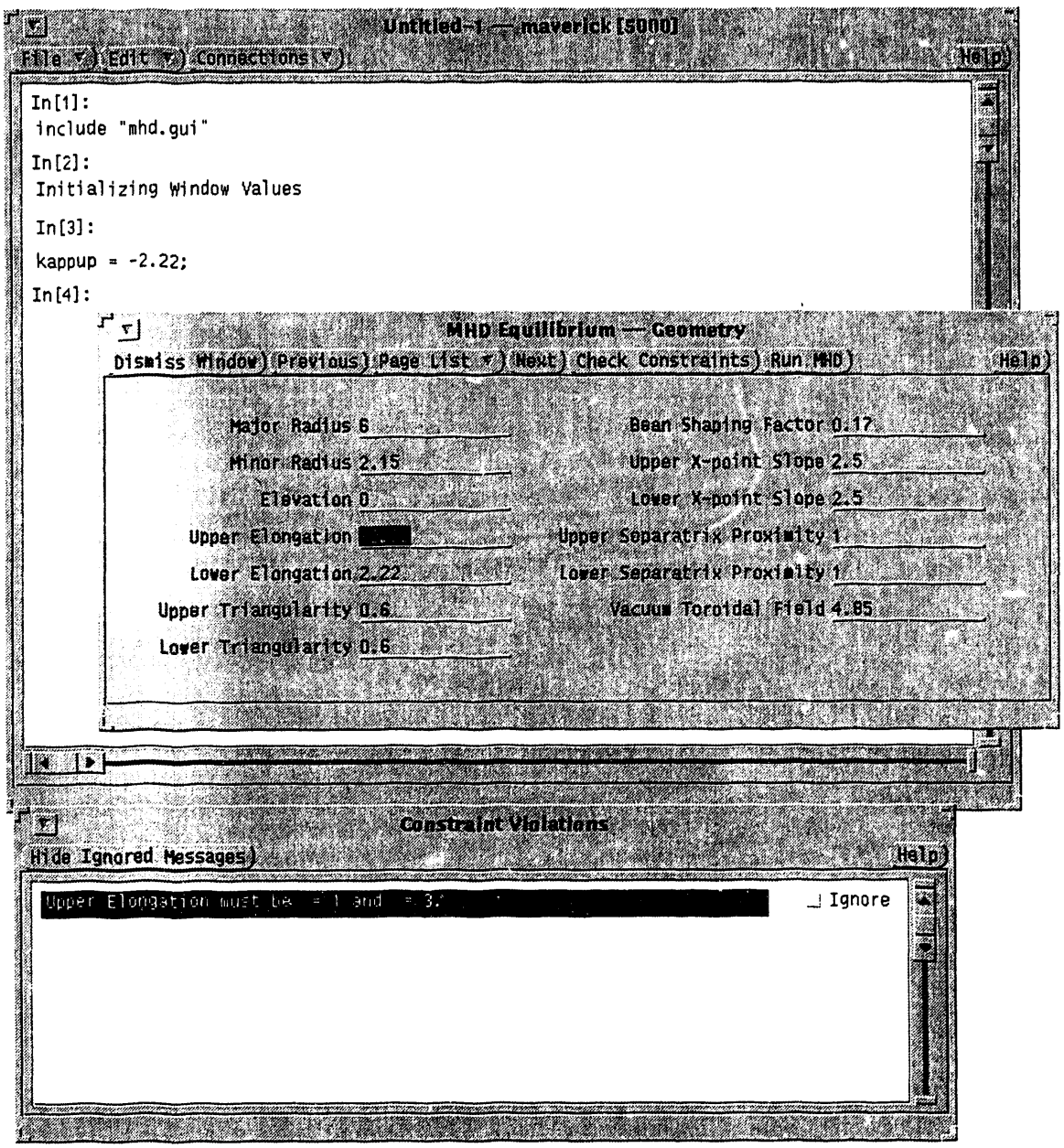

Figure 4: Terminal window, GUI defined by file mhd.gui, *and the Constraints Violations window. The user clicked on the error message which automatically selected the offending entry field.*

possibly violates constraints. Since the user is to be notified as soon as possible when errors occur, the GUI also sends the shell a command to check the constraints each time it changes the state of the system. The shell checks the constraints and executes the associated action routines for every constraint that is violated. In the prototype, the shell sends the GUI a message whenever a violation occurs, and the GUI displays a message to the user in a *Constraint Violations* window. Figure 4 shows a *Constraint Violations*

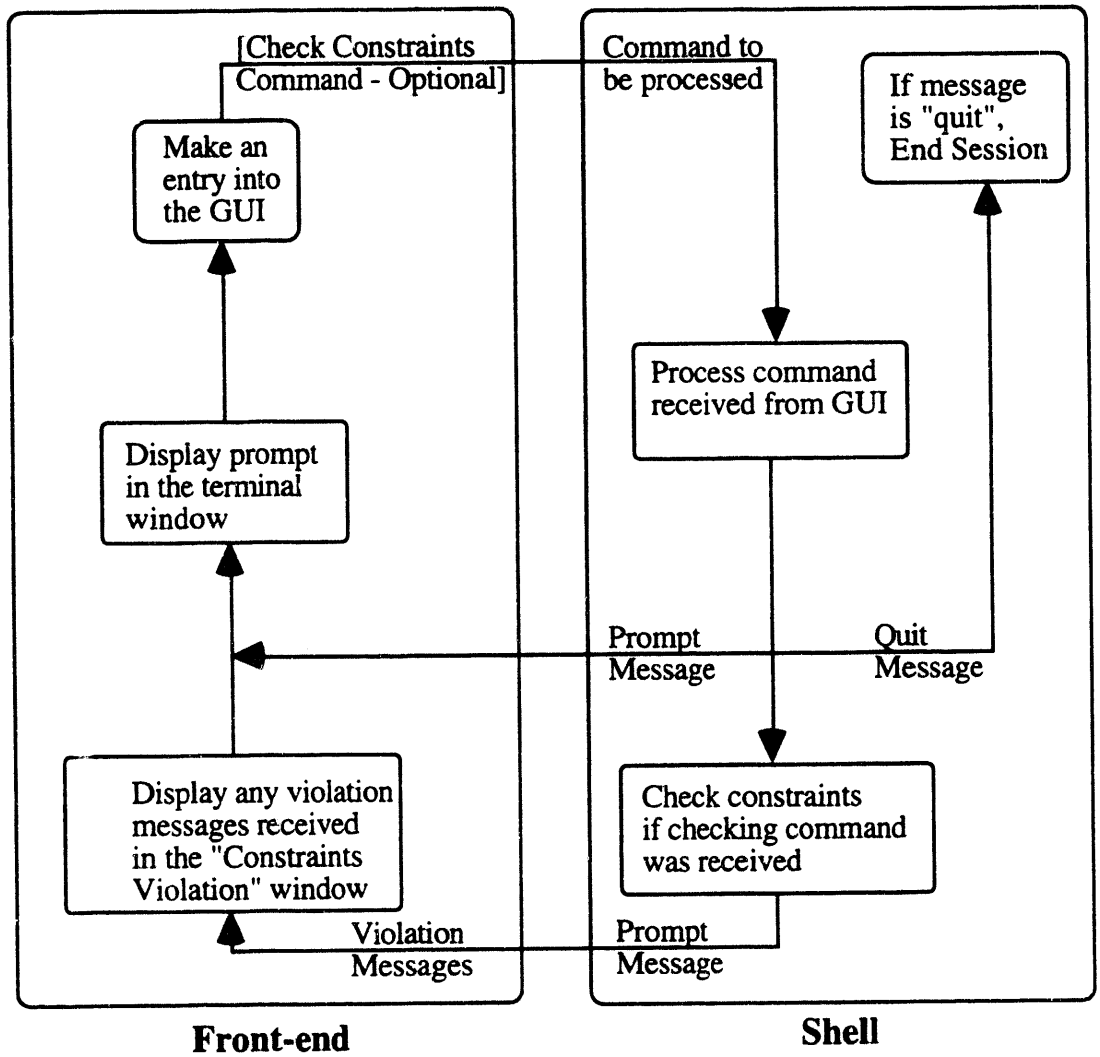

Figure 5: Flow of control diagram depicting typical use.

window and also the window's highly desireable feature: the front-end automatically selects the offending entry field when the user highlights an error message. The described flow of control, as depicted in Figure 5, represents a typical interaction between a user and the prototype system.

3.2 The GUI Creation System

Before a user can use SUPERCODE, an expert GUI designer must create the file that dynamically configures and displays the friendly interface. The GUI creation system enables expert programmer/SUPERCODE designers to easily create GUI specification files. Since the shell interpreter implements a large subset of C++, the GUI creation system is also implemented in C++ with GUI component classes added to the shell class library.

The prototype currently provides the GUI developer with the ability to create and display `InputWindows`. An `InputWindow` is a window that has a main menu for executing actions that are common to all `InputWindows`. In addition, it contains an arbitrary number of `Pages`. A `Page` is a box that resides within an `InputWindow` that can contain other GUI components. One `Page` at a time is displayed in an `InputWindow`. The GUI components include, for instance, a `RealEntryField` for representing real variables, a `Button` for executing actions, a `Label` for headings and instructions, and a pull-down `Menu` for making selections. Some SUPERCODE-specific GUI items were also created.

The GUI construction process proceeds as follows: A GUI developer creates a file with the statements that produce a specific interface. This file, when read and processed by the shell, creates a simple interface by sending the interface information to the front-end. The front-end uses this information to generate and display the appropriate GUI. Figure 6 shows the contents of file `fam.gui` and the resulting simple interface is displayed in Figure 7.

```
1    Real grandpa = 59, mother = 30;
2    InputWindow famWindow("Family Members");
3    Page page1("Page 1", 1);
4    famWindow.addPage(page1);
5    RealEntryField gramps("GrandPa","grandpa",1,1);
6    page1.addRealEntryField(gramps);
7    RealEntryField ma("Mother","mother",1,2);
8    page1.addRealEntryField(ma);
9    famWindow.show();
```

Figure 6: Code contained in GUI specification file `fam.gui`. *Note that line numbers are used only for reference purposes.*

Line 1 declares variables `grandpa` and `mother` of type `Real`, initializing them to `59` and `30` respectively. Line 2 creates an `InputWindow` object, `famWindow`, the title of which is `Family Members`. Line 3 creates a `Page` object, `page1`. Its title is `Page 1` and its

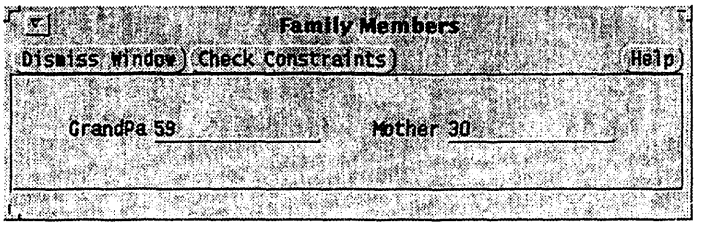

Figure 7: The GUI the front-end displayed when the shell read and processed file, `fam.gui`.

page number is `1`. The page number is used for identification purposes. Line 4 adds `page1` to `famWindow`. Line 5 creates a `RealEntryField` object, `gramps`, and its label is `GrandPa`. The string parameter "`grandpa`" indicates the identifier to which an entered value is assigned, and the last two parameters, `1` and `1`, are the row and column position where `gramps` will be displayed when it is added to a `Page` object. Line 6 adds `gramps` to `page1`. Lines 7 and 8 repeats the process just described for lines 5 and 6 for a `RealEntryField` labeled `Mother`. The shell builds GUI objects as it executes lines 1 through 8. When the shell executes line 9, it sends the interface information to the front-end. The front-end uses the information received from the shell to build the appropriate interface.

A major advantage of this system is that the GUI developer can write and save files that create specific interfaces for use by new users. These interfaces can be much more complicated than the simple example presented. `InputWindows` can consist of an arbitrary number of `Pages` with arbitrary numbers of `RealEntryFields`, `Buttons`, `Menus`, etc.

3.3 Constraint Language Syntax

In addition to creating the GUI, the designer may define constraints to provide error detection and handling for the end-user. An easy-to-use constraint definition syntax is a high priority issue with GUI designers. It is important that the syntax be both intuitive and concise if we are to entice GUI designers to use the system. The syntax should be intuitive so that GUI designers can easily understand how to define a constraint. Since it is reasonable that over a hundred constraints may need to be defined, the syntax should be as concise as possible. A syntax that reduces the required characters by only 20% per definition can save the GUI designer hundreds of keystrokes in one GUI specification file.

Standard C++ exception handling was initially considered but rejected because it provides language-level, as opposed to user-controlled, exception handling, and it is designed to deal with many situations that are not applicable to our system. We decided it was better to design a system tailored to our project. Two of the syntactical forms considered are presented:

```
require (<expr>)
{
  <statement-list>
}
```

and

```
ifnot (<expr>)
{
  <statement-list>
}
```

These forms were rejected because they have an if-statement like appearance. If-statements only execute as program control passes through them. Our intention was to define *persistent* constraints that act as guards against errors. Also, since the intent is to create a constraint, a declarative syntax is more intuitive. We chose the following syntax for the prototype:

```
Constr <identifier> (<expr>) {
  <statement-list>
}
```

`Constr` is the name of a class within the shell that implements constraint checking and error handling. The unique name of the constraint is `<identifier>`. The condition clause used for error detection is `(<expr>)`. It consists of C++ code that generates a non-zero value if the constraint is satisfied and a zero value if a violation occurs. The action statements used to handle the error if a violation occurs is `<statement-list>`. Note that although the constraint identifiers are unique, it is possible for multiple constraints to have identical `<expr>`'s and/or `<statement-list>`'s. The prototype has not been equipped to optimize this situation. If duplicate `<expr>`'s or `<statement-list>`'s exist, these duplicates are entered into the system.

Suppose a GUI designer wishes to require variable `grandpa` to be greater than variable `mother` and, if violated, wants to user notified. The GUI designer would write the following statement:

```
Constr _grandpa (grandpa > mother){
  Violation("Grandpa should be older than mother.");
}
```

The constraint identifier is `_grandpa`, the boolean expression `(grandpa > mother)` is the condition clause for detecting the error, and `Violation("Grandpa should be older than mother.")` is a subroutine call for handling the error. When `Violation` executes, it sends the string parameter to the front-end to be displayed in the Constraints Violation window.

3.4 The Preprocessor

Neither the C++ language nor the subset of C++ implemented by the shell supports class object declarations with the chosen syntactical form and the kind of persistent action we require. Therefore, we built a preprocessor to convert the constraint definition syntax to a C++ form implemented by the shell.

Because of the characteristics of the shell and the preprocessor, there must be a space between the constraint identifier and the condition clause. Moreover, the preprocessor employs a naming convention such that the `Constr` identifer must be the name of the variable constrained with a prepended underscore. This naming convention enables the preprocessor to automatically add a necessary second parameter to the `Violation` function call.[1] However, it is possible that a designer will place a constraint on an array element. Because the shell implements arrays with the FORTRAN syntax and the preprocessor was built using LEX [22] (which performs only lexical analysis), the preprocessor needs the space to distinguish between `Constr` identifier and the condition clause. Despite the imposed naming convention and necessary space, the syntax is concise and easy to understand.

In addition, the preprocessor is effective in reducing the number of characters a GUI designer must code to define a constraint. For example, the preprocessor converts

```
Constr _grandpa (grandpa > mother){
  Violation("Grandpa should be older than mother.");
}
```

[1] When `Violation` sends an error message to the front-end to be displayed, the name of an identifier with which it can be associated must also be sent. This is necessary to implement a highly desireable feature: automatic selection of the offending entry field when the user clicks on the error message. This feature requires the string representation of the identifier to be a parameter of the `Violation` function call.

to

```
Void c_grandpa()
{
  Constr::returnValue = (grandpa > mother) ? 1 : 0;
}

Void a_grandpa()
{
  Violation("Grandpa should be older than mother.",
            "grandpa");
}

Constr _grandpa(c_grandpa, a_grandpa);
```

Notice that for this simple example, the C++ form requires 174 characters of code while the code the GUI designer must write and preprocess is only 84 characters. This savings of over 50% is significant, especially when many constraints are defined. The usability of the syntax and the potential reduction in coding for the GUI designer makes the preprocessor a desirable tool.

3.5 Implementation

The `Constr` class definition and those for the related classes, `ConstrNode`, `Table`, and `TableEntry`, are presented in Figure 8.

3.5.1 Constraint Creation

Notice in Figure 8 that the `Constr` constructor takes as parameters the pointers to the error detector function and the error handler function previously defined. The constructor creates a `ConstrNode` object setting its member variables `Error` and `Handler` to the pointers to the error detector and the error handler functions respectively. The pointer to this `ConstrNode` object is then assigned to the `Constr` member variable `constrNodePtr`. It is also stored in a list called `constraintList`, a static member of class `Constr`. It is used for the naive checking scheme.

```
class Constr {
public:
  Constr(void (*conditional)(), void (*action)());
  ~Constr();
  static Table hashTable;
  static void check();
  static void minCheck(String identifier);
private:
  static int returnValue;
  static ConstrNode *constraintList;
  ConstrNode *constrNodePtr;
  void addToTable(String identifier);
};

class ConstrNode {
  friend class Constr;
public:
  ConstrNode(void (*conditional)(), void (*action)());
  ~ConstrNode();
private:
  void (*Error)();
  void (*Handler)();
  ConstrNode *nextNode;
};

class Table {
  friend void Constr::minCheck(String identifier);
  friend void Constr::addToTable(String identifier);
public:
  Table();
  ~Table();
  TableEntry *entry[MAXTABLESIZE];
};

class TableEntry {
  friend void Constr::minCheck(String identifier);
  friend void Constr::addToTable(String identifier);
public:
  TableEntry(String identifier, ConstrNode *node);
  ~TableEntry();
  String id;
  ConstrNode *constrList;
};
```

Figure 8: Class definitions for classes `Constr`, `ConstrNode`, `Table`, *and* `TableEntry`.

3.5.2 The Naive Checking Scheme

The naive checking scheme is straightforward; it simply checks every defined constraint. Because every constraint is checked, it is very robust. Every `InputWindow` the GUI displays is equipped with a *Check Constraints* button to allow the user the option of checking all defined constraints at any time. This scheme requires no overhead to start the checks, and all violations are detected as their constraints are evaluated. However, this scheme makes no attempt to determine whether or not a constraint is related to the current system change.

Recall that the `Constr` static class member, `constraintList`, is a list of pointers to all of the `ConstrNode` objects containing pointers to the error detector and error handler functions. The `Constr` static member function, `check`, traverses this list in the following manner: For each `ConstrNode` object pointed to in the list, starting at the front, `check` executes the error detector function and, if an error is detected, executes the error handler function.

The constraints are checked in this manner every time the shell receives the `check` function call from the GUI. Specifically, the GUI sends the shell `Constr::check()`. Since the user must be notified of the error as soon as possible, the GUI sends the check command every time a GUI user event results in a system change (i.e., the user enters a new value for a variable). However, there can be hundreds of constraints defined and few, if any, of the constraints may be related to a single change of value. Checking every constraint can result in excessive overhead. A minimum evaluation scheme is implemented to reduce the time spent trying to detect errors.

3.5.3 The Minimum Evaluation Scheme

The minimum evaluation scheme reduces the time spent trying to detect errors by checking only those constraints related to a single value change. This requires some overhead to decide what the appropriate set of constraints is. However, there are potential savings if this decision cost is relatively low compared to the constraint evaluation time and the number of constraints evaluated are reduced.

To check only those constraints related to a single value change, a method of mapping an identifier to the set of related constraints is needed. The preprocessor scans the

condition clause of the constraint declaration and generates a read set consisting of all the identifiers in the conditional. For example, in the constraint declaration

```
Constr _grandpa (grandpa > mother){
  Violation v("Grandpa should be older than mother.");
}
```

`grandpa` and `mother` are the identifiers which constitute the read set of the condition clause, `(grandpa > mother)`. We use a hash table to store the identifiers with a list of their relative constraints. The preprocessor automatically adds the command to the output file that enters the information into the hash table. For this example, the preprocessor would add the following commands:

```
_grandpa.addToTable("grandpa");
_grandpa.addToTable("mother");
```

The `addToTable` function call is a private member function of class `Constr`. This function creates a `TableEntry` object whose `String` member, `id`, is assigned the `String` parameter of the `addToTable` function call. In addition, it adds the calling `Constr` object's `constrNodePtr` to the `TableEntry` object's list of `ConstrNode` pointers, `constrList`. Next, `addToTable` enters the `TableEntry` object into the hash table using Holub's [23] hashing function.

As user-inputs are made, if the value of `grandpa` or `mother` changes, instead of sending the shell the command to execute the `Constr` static member function `check`, the GUI sends the command to execute the `Constr` static member function `minCheck`. Specifically, the GUI sends the shell `Constr::minCheck("grandpa")` or `Constr::minCheck("mother")` if the value of `grandpa` or `mother` changed respectively. When the `minCheck` function is called, it applies the hash function to its `String` parameter to find the appropriate entry slot of the hash table. Next, it compares its `String` parameter to the `TableEntry` object's `String` member, `id`. If `id` does not match the String parameter, the next `TableEntry` in `Table::entry` is examined. When the correct `TableEntry` object is found, `minCheck` executes the error detector function and, if violated, the corresponding error handler function, for every `ConstrNode` pointed to in the `TableEntry` object's member, `constrList`. In this manner, when the value of `grandpa` or `mother` changes, only the related condition clause `(grandpa > mother)` is checked for possible violation. Other constraints that do not involve `grandpa` or `mother` are not evaluated.

The generated read sets help the minimum evaluation scheme determine which constraints to execute when an identifier's value changes. However, what happens if the expression contains a function call? Because a function call can contain hidden reads, the read set of identifiers generated by scanning the condition clause may not be sufficient to determine which constraints need to be checked. If the system always had access to the source code of the functions used in the condition clauses, symbolic execution could be used to generate the complete read sets. However, the source code of these functions may reside within SUPERCODE's system, and some functions may be compiled. Either way, the preprocessor that creates the read sets does not have access to the source code.

To handle this situation, a *wild card* list can be generated containing all of the constraints for which the read set is indeterminate. When a function call is encountered in a condition clause, the corresponding constraint is added to the list. The constraints in the wild card list are checked every time in addition to the constraints known to be associated with the system change. Although more constraints are checked than is probably necessary, this is a simple solution to overlooking a constraint that should be checked.

However, functions often do not contain any hidden reads, and a method should be provided to avoid adding constraints to the wild card list when this is the case. For example, there are no hidden reads within the trigonometric functions: `sin(x)`, `cos(x)`, `tan(x)`, etc. A compiler directive can provide a tag indicating a function is "OK" when it is known to produce no side effects. Expressions containing functions that are tagged as OK may be processed in the original read set manner. This would reduce the number of constraints contained in the wild card list that must be evaluated after every system change.

The prototype does not provide a wild card list when functions are part of the expressions of the constraints. At this time, the prototype properly handles only those constraints strictly involving identifiers. Extending the system to properly handle error clauses containing function calls is a topic for future work.

Chapter 4

Results

There are three areas in which the system can succeed or fail: usability of the GUI creation system, run-time optimization of the minimum evaluation scheme, and the usability of the end-user interface. GUI designer testing is used to determine the usability of the GUI creation system. To determine if the minimum evaluation scheme is a success, run-time comparisons between the minimum evaluation scheme and the naive checking scheme are made for both extreme and representative cases. Finally, although human-computer interaction guidelines are examined, we ultimately determine the success of the end-user environment by analyzing end-user feedback.

4.1 GUI Creation System Success

Before an end-user can utilize one of these robust and helpful interfaces, an expert GUI designer must design it. To persuade the GUI designer to use the GUI creation system, we made the GUI creation process described in Chapter 3 as easy and succinct a process as possible.

Whether or not the GUI creation process is easy and succinct enough to promote use by GUI designers is difficult to determine. The GUI creation system environment is a programming environment, and a good programming language is both readable and writable [24]. A readable programming language is one whose meaning is easily extracted from the syntax. A programming language is writable if it is concise and easy to generate. A readable programming language that is verbose is not writable just as a writable

programming language that is cryptic is not readable. We assessed the readability and writability of the GUI creation system's programmer interface by asking three expert GUI designers to read the user's manual, create a GUI, and define a constraint.

The designers found the constraint language syntax very usable. All of the designers felt that more friendly GUI's created by designers would encourage distribution to end-users. They also felt creating the GUI specification files is worth the extra effort to promote this distribution. One designer stated, "People may want to customize the input, etc. depending on the particular problems they are running. A GUI creation system will be very useful." One designer liked it so much that he requested porting it to work with codes created using the Basis System. The designers found the GUI creation system to be a valuable tool and would use it themselves.

The designers also had some useful comments concerning the system. One designer said, "My first reaction was that the `Constr` should be executed if the specified condition is True (opposite of the way it is set up now), but this is a small point." The logic is set up to execute the action routine if the condition clause evaluates to false. It was done this way because we view defining a constraint as placing restrictions on user inputs, and the present implementation follows that logic. As the user said, it is a small point, but we will keep this in mind as more designers use the system. Another user commented, "It is simple, so I like it in that sense. It doesn't look like C++ and so it seems a bit unnatural. I've often used preprocessors to hide "ugliness" of a C/C++/Mppl implementation, though." None of the designers could suggest a more appropriate syntax, and all of them found it easy to use. Based on user testing we conclude that the GUI creation system was successful.

4.2 Effectiveness of the Minimum Evaluation Scheme

The end-user environment not only has to be easy to use, but it must also have a response time that promotes use. The front-end is meant to augment the command line interface, and although somewhat dependent on the skill of the GUI designer, the end-user GUI's provide a much friendlier interface. However, the response time of the system can greatly affect whether or not a casual user will indeed use the system. Aside from the complexity of the error tests of the constraints defined, the GUI designer has no control over the response time.

To measure and optimize the response time, we are interested in how long it takes to complete these steps:

1. Sending a system changing command from the front-end to the shell.
2. Processing the commands.
3. Checking the constraints.
4. Sending response from the shell to the front-end.
5. Processing the shell's response.

To optimize this response time we must identify the parts of the process path that significantly impact the response time and make them as efficient as possible. Because there is little means to reduce the network communication time between the shell and front-end,[1] parts (1) and (4) offer little opportunity for optimizing response time. Part (2) depends on the efficiency of the shell and is beyond the scope of this prototype. Part (5) requires the GUI to, at most, construct and display error messages in the *Constraint Violations* window and a prompt in the terminal window. This is dependent on the OI class library, so we can have little impact on this part. However, there is a potential for optimizing part (3). This is why we implemented the minimum evaluation scheme.

4.2.1 Common Experiment Factors

There are some basic factors common to all of the following experiments. Times for the minimum evaluation scheme were obtained using the following procedure. Recall that the minimum evaluation scheme checks constraints by executing the class `Constr` static member function `minCheck`. The `minCheck` function takes a string as an argument, and its execution is timed for `N` (typically `N = 1000`) iterations. Because we are interested in only the time required to execute `minCheck`, the time required to call `minCheck` (i.e., call to an empty function that with a string argument) is subtracted, and this difference is normalized by dividing by `N`:

$$1\ minCheck\ execution = \frac{N * \left(minCheck(string) - emptyFunct(string)\right)}{N}$$

[1] Reducing network communication time can be accomplished by having the front-end and the kernel residing on the same local area network will reduce.

Times for the naive checking were obtained using a similar procedure with one difference. The class `Constr` static member function `check` which checks the constraint for the naive checking scheme takes no arguments while the `minCheck` has a string argument. Therefore the subtracted time is the time required to execute a `N` iteration loop containing a call to an empty function that passes no arguments:

$$1\ check\ execution = \frac{N * (check() - emptyFunct())}{N}$$

These times reflect the average time required to execute their respective checking schemes. Also, since the time required to execute the error handler functions is identical between the minimum evaluation scheme and the naive checking scheme, and since we are only interested in their relative run-times, the experiment assumes that constraints are always satisfied (i.e., no action code is called). The experiments presented are the "Best Case," "Worst Case," "MHD Equilibrium" example, and "Average SUPERCODE" example.

4.2.2 Best Case

The best possible case, in terms of run-time, occurs when all of the constraints are independent and there are no collisions in the hash table used to implement the minimum evaluation scheme. No collisions in the hash table means there is at most one identifier associated with any one hash table slot. This situation has the fastest possible lookup for the minimum evaluation scheme. When the constraints are independent, changing any one identifier requires that exactly one constraint needs to be checked. The naive scheme requires no lookup time, but it has to check every constraint. We will show that the minimum evaluation scheme outperforms the naive scheme except for a very small number of constraints.

Setup The experiment was set up in the following manner. Hash table collisions were avoided by selecting appropriate value identifiers. To ensure that all constraint checks take the same time, all constraint condition clauses were structurally of the form (identifier >= 0). This experiment measures the run-times for both the minimum evaluation scheme and the naive checking scheme.

The expected cost to run the naive checking scheme is

$$T_{naive}(n) = n * CheckTime$$

where n is the number of constraints defined, and *CheckTime* is the time it takes to run a constraint function. This indicates that the naive checking scheme run-time will increase linearly with the number of constraints defined. The expected cost of the minimum evaluation scheme is

$$T_{best}(n) = DecisionTime + CheckTime$$

where *DecisionTime* is the time required to find the check list associated with a particular identifier, and *CheckTime* reflects the fact that there is always only one constraint checked. Since we implemented the minimum evaluation scheme with a hash table (with overflow chaining), we see the decision time can be further broken down into

$$DecisionTime = (Collisions + 1) * (ChainAccess + StringCompare)$$

where *ChainAccess* is the time required to access a hash table slot, and *StringCompare* is the time required to determine if the slot contains the desired entry. *Collisions* is the number of collisions occurring at the hash table slot, and its value is in the range $[0, n - 1]$. Note that one must be added to *Collisions* because one *ChainAccess* operation and one *StringCompare* operation must be performed every time the minimum evaluation scheme determines if it has the correct list of constraints to check. This is true even when there are no collisions. However, for this experiment we have chosen identifiers that do not collide in the hash table. Therefore, *DecisionTime* becomes

$$DecisionTime = ChainAccess + StringCompare$$

indicating a constant run-time for the minimum evaluation scheme. Because of this constant run-time, the relationship

$$DecisionTime < (n - 1) * CheckTime$$

is true for some n.

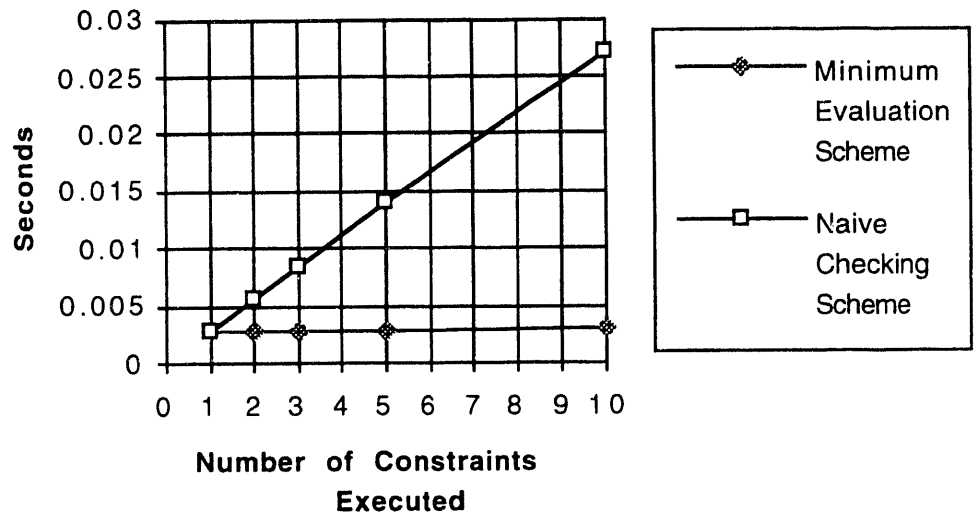

Figure 9: Best case run-time results for the minimum evaluation scheme and the naive checking scheme.

Evaluation Figure 9 details the run-times for this best case scenario. As expected, the minimum evaluation scheme has an almost constant run-time regardless of the number of constraints defined. Also, the naive checking scheme run-time grows linearly with the number of defined constraints. Notice that both schemes appear to have identical run-times for one constraint. Actually, the minimum evaluation scheme has a slightly higher run-time than the naive checking scheme. This small difference shows up in Figure 10. The slightly higher run-time of the minimum evaluation scheme is consistent with the predicted additional decision cost associated with the hash table implementation.

In the prototype, the decision cost was approximately 0.00003 seconds while the single constraint cost was approximately 0.00284 seconds. So, the expected cost of the naive scheme for n constraints is

$$T_{naive}(n) = n * (0.00284\ seconds)$$

and the expected best case time for the minimum evaluation scheme is

$$T_{best}(n) = 0.00003\ seconds + 0.00284\ seconds$$

Therefore, the minimum evaluation scheme in the prototype breaks even at 1.011 constraints, outperforming the naive checking scheme when as few as two constraints are

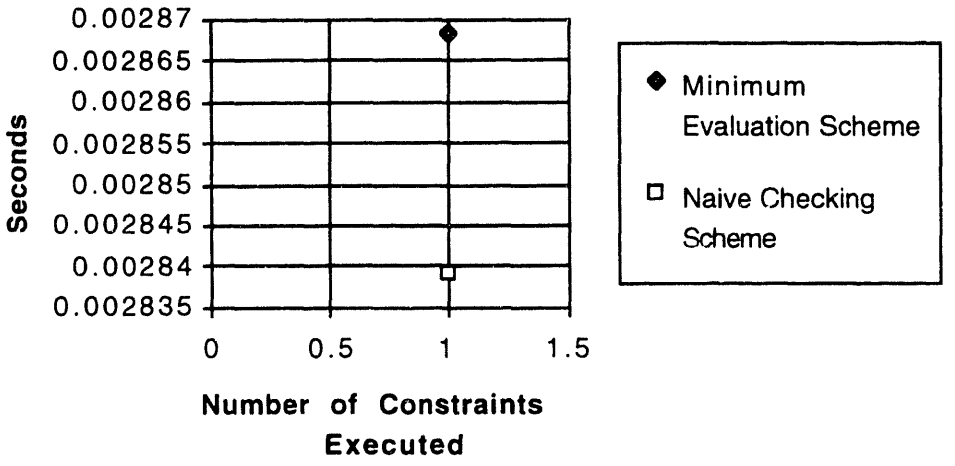

Figure 10: Blowup of best case run-times data point for one constraint.

defined. This is because the decision cost has a much lower execution time relative to the constraint check even for very simple constraints. The *ChainAccess* and *StringCompare* are executed in the shell by a compiled class `List` operation and a compiled class `String` operation, respectively, while the constraint check is executed as an interpreted function. The interpreted nature of the shell skews the results. If the chain access and string compare cost was relatively higher (as it would be if the constraint checks were compiled functions), then the break-even point would move higher. To find out how much higher, we added functions required to check the constraints in the best case to the system and compiled it. We found that the decision cost in the compiled version is actually higher than the time required to execute the simple error test function. As a result, the minimum evaluation scheme does not outperform the naive scheme until at least five constraints are defined (see Figure 11). However, the prototype is not a compiled system, so we will not concern ourselves with what happens in a compiled system any further. The rest of this paper will address only the existing interpreted environment.

Significance In this best case, the minimum evaluation scheme maintains a near constant run-time while the naive scheme run-time increases almost linearly with the number of constraints. The minimum evaluation scheme is a success in the best case by outperforming the naive checking scheme when two or more constraints are defined in the interpreted system. This holds true in compiled systems as well. However, the break-even point is quickly reached at only five constraints.

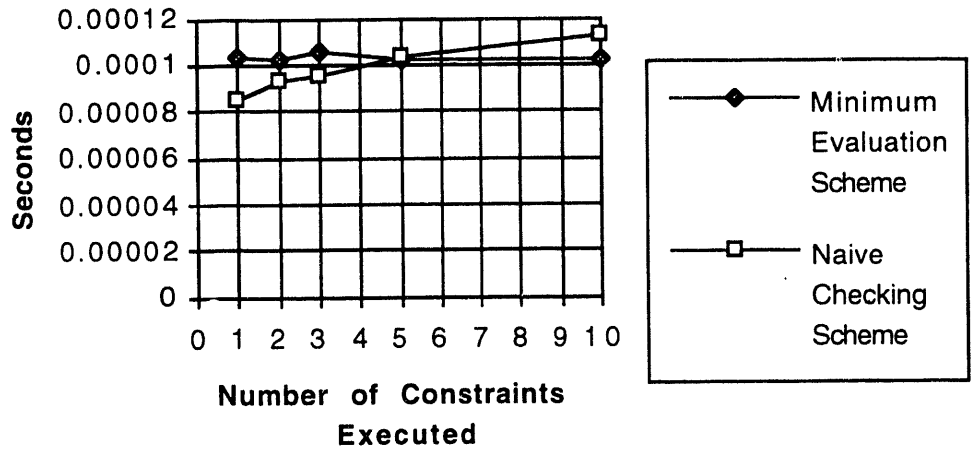

Figure 11: Best case run-time results for the minimum evaluation scheme and the naive checking scheme in a fully compiled system.

4.2.3 Worst Case

The worst possible case, in terms of run-time, for the minimum evaluation scheme occurs when all of the constraints are interdependent, when there is a 100% collision rate in the hash table used to implement the minimum evaluation scheme, and when the last member in the collision chain is the one that is accessed. When the constraints are interdependent, changing any one identifier requires that all of the constraints need to be checked. In addition, the error tests involve all of the identifiers and are both more complicated and slower to execute. A 100% collision rate in the hash table means that all of the identifiers associate with the same hash table slot. This situation has the slowest possible lookup for the minimum evaluation scheme when accessing the last member in the collision chain. We will show that the minimum evaluation scheme for this Worst Case is only slightly slower than the naive checking scheme.

Setup Hash table collisions were produced by selecting appropriate value identifiers. To ensure that all constraint checks take the same time, all constraint condition clauses were of the form $(identifier_1 + identifier_2 + \ldots + identifier_n < 100)$ for n value identifiers. This experiment measures the run-times for both the minimum evaluation scheme and the naive checking scheme.

Both the naive checking scheme and the minimum evaluation scheme must check every constraint. However, the minimum evaluation scheme must also find the correct list of constraints to check in the hash table collision chain. Therefore, the naive scheme is expected to have the smaller run-time.

More specifically, the run-time for the naive checking scheme is

$$T_{naive}(n) = CheckTime_1 + CheckTime_2 + \ldots + CheckTime_n,$$

and the expected cost to run the minimum evaluation scheme is

$$T_{worst}(n) = DecisionTime + CheckTime_1 + CheckTime_2 + \ldots + CheckTime_n.$$

In both of these equations, n is the number of constraints defined and $CheckTime_1 + CheckTime_2 + \ldots + CheckTime_n$ is the time it takes to check each of the n constraints. It is important to realize that as the number of constraints defined increases, the complexity of the condition clauses also increases. This means that $CheckTime_1$ when $n = 1$ is going to execute faster than $CheckTime_1$ when $n = 10$. *DecisionTime*, found only in equation $T_{worst}(n)$, is the time the minimum evaluation scheme requires to find the check list associated with a particular identifier. Because *DecisionTime* is

$$DecisionTime = (Collisions + 1) * (ChainAccess + StringCompare)$$

and because the number of collisions increases with the number of constraints defined, the decision cost also increases as the number of defined constraints increases.

Evaluation Figure 12 represents the run-times for this Worst Case scenario. The graph indicates a slight curve as n increases. This is due to the previously stated fact that condition clauses of each constraint increases in complexity, and therefore execution time increases as n increases. The graph also indicates that the minimum evaluation scheme and the naive checking scheme have the same performance. Actually, the minimum evaluation scheme performs slightly worse. The difference between the two checking schemes did not show up in Figure 12 because the additional decision cost that minimum evaluation scheme incurs is very small relative to the cost of the constraint checking cost. A blowup of Figure 12 (see Figure 13) of the run-times for one and two constraints reveals this very small difference.

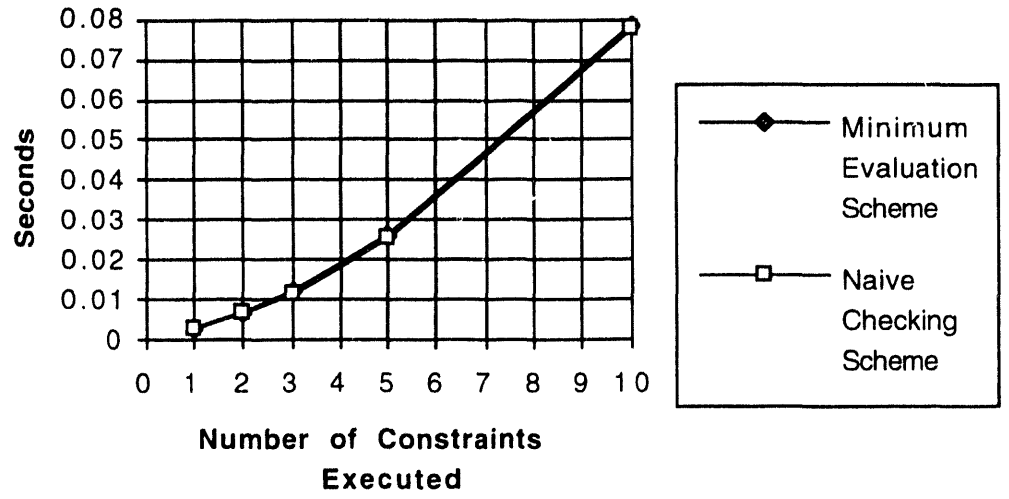

Figure 12: Worst Case run-time results for the minimum evaluation scheme and the run-time for the naive checking scheme.

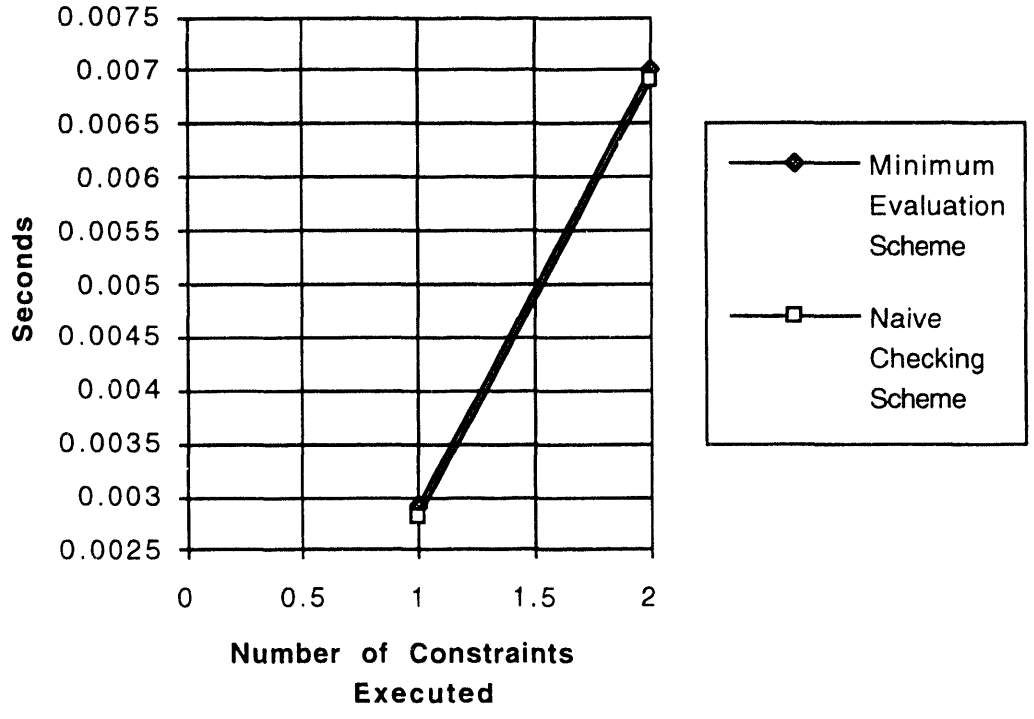

Figure 13: Blowup of worst case run-times for one and two constraints.

Significance In this worst case, the minimum evaluation scheme's run-time is only slightly more than the naive scheme's. Even though the minimum evaluation scheme did not outperform the naive checking scheme in this case, this is a very extreme case and

simply will not occur in any SUPERCODE run. In addition, the small decision cost and the experiment results presented demonstrate that it is not the chosen implementation (i.e., using a hash table) that will determine whether or not the minimum evaluation scheme significantly outperforms the naive checking scheme, but rather the percentage of the total number of constraints defined that the minimum evaluation scheme must check.

4.2.4 MHD Equilibrium

The results from the two previous experiments provide good lower and upper bounds for the minimum evaluation scheme, but represent unrealistic scenarios. Analyzing the SUPERCODE's MHD Equilibrium case shows the better performance of the minimum evaluation scheme in an actual case.

Setup The system identifiers for the MHD Equilibrium case were used and appropriate constraints were defined. Because of the nature of this case, the condition clauses are not of the same structural form, and the number of constraints that are executed by the minimum evaluation scheme varies from one to three. The hash function worked well for this case, and all of the identifiers in this example have unique hash values. Therefore, to obtain a representative set of data, run-times were gathered for a set of identifiers (`plascur`, `beansh`, `nrho`, `ctroy`, `rmajor`) that represented each condition clause's structural form and each number of constraints executed. Run-times were measured for both the minimum evaluation scheme and the checking scheme.

Since the condition clauses are not necessarily of the same structural form, the run-time for the naive checking scheme is

$$T_{naive}(n) = CheckTime_1 + CheckTime_2 + \ldots + CheckTime_n$$

where $n = 30$ constraints are defined in this example. The expected cost to run the minimum evaluation scheme is

$$T_{mhd}(n) = DecisionTime + CheckTime_i + CheckTime_j + \ldots + CheckTime_r$$

where $CheckTime_i$, $CheckTime_j$, . . ., $CheckTime_r$ are the times to check each of the related constraints.

Case	% of Constraints Changed	Executed [%]	Time [sec]
Naive	N/A	100	0.06024
Set 1			
Minimum Evaluation Scheme	plascur	3.33	0.01184
	beansh	3.33	0.02478
	nrho	3.33	0.02257
	ctroy	6.67	0.02321
	rmajor	10	0.08300
Set 2			
Minimum Evaluation Scheme			
using Reference Variables	plascurRef	3.33	0.00287
	beanshRef	3.33	0.00397
	nrhoRef	3.33	0.00430
	ctroyRef	6.67	0.00595
	rmajorRef	10	0.01157

Table 1: MHD Equilibrium run-times results for a descriptive set of identifiers (plascur, beansh, ctroy, nrho, and rmajor) *when using both static class vairables and reference variables. Run-time for the naive checking scheme is also presented.*

Evaluation Table 1 lists the run-times for this MHD Equilibrium case. It is obvious that the naive checking scheme runs significantly slower than the minimum evaluation scheme. Notice, however, that the first set of minimum evaluation scheme run-times are higher than expected. Recall that the identifier, plascur, uniquely hashed into the hash table and had a condition clause of the form (identifier >= 0). Knowing this, we expected a run-time close to 0.00287. Yet, we got a run-time of 0.01184. Why is this run-time over four times that of the expected value?

The difference in the run-times is due to the difference in the identifiers and how the shell interpreter treats them. The identifiers used in the best case are global identifiers while the identifiers used in the MHD Equilibrium example are static class identifiers. In C++, a static class identifier must be referenced using a specific syntax: ClassName::identifier. Otherwise, an undefined-identifier error occurs. To avoid referencing static class identifiers by the longer syntax, the shell must perform a runtime search of all of the classes for static class identifiers that match the referenced identifier before producing the undefined identifier error. If the search finds a unique match, that static class identifier is

assumed to be the identifier referenced. If a unique match is not found, an error is produced. The increased time reflects this additional search time.

Fortunately, this extra time can be avoided without typing the longer syntax by using reference identifiers. If a reference identifier is declared and set equal to a static class variable, this reference variable can be used to access the static class variable with the comparable speed of the global identifiers. The second set of minimum evaluation scheme run-times in Table 1 list the run-times for the same set of identifiers when reference variables are used. Notice that the run-times are close to the expected. The run-time of the minimum evaluation scheme when the constraint for `plascur` is checked is 0.00287. This is the expected run-time. The run-time of the minimum evaluation scheme when the three constraints for `rmajor` are checked is 0.01157. This is more than 3 * 0.00287 = 0.00861, but this is reasonable due to the more complicated condition clauses that were executed. The condition clauses for `rmajor` are `((rmajorRef > 0) && (rmajorRef >= rminorRef))`, `(rmajorRef > 2 * rminorRef)`, and `(rminorRef > 0)`. Using the reference identifiers greatly reduces the time required to access the static class identifiers and therefore reduces the execution time of the minimum evaluation scheme.

Significance In this MHD Equilibrium example, the run-times, when reference identifiers are not used, are over four times greater than when reference variables are used. The minimum evaluation scheme has a maximum run-time of approximately 0.08300 which is significantly reduced to approximately 0.01157 by using reference identifiers. Because of this time reduction, reference identifiers should be used when defining constraints on static class identifiers. Also, the run-time for the naive checking scheme for the 30 constraints is approximately 0.60244 without reference identifiers and 0.1 with reference identifiers. The results of this MHD Equilibrium experiment indicate the minimum evaluation scheme is a success because it significantly outperforms the naive checking scheme.

4.2.5 Average SUPERCODE Systems Runs

Although the MHD Equilibrium experiment represents a real-life scenario, it is a relatively simple case. Larger SUPERCODE systems runs involve as many as 150 total constraints defined, and the percentage of the constraints related to any system change can range between 1%-5%. Analyzing cases with the average SUPERCODE systems run characteristics demonstrates the performance of the minimum evaluation scheme for the average SUPERCODE case.

Setup Three cases with 150 defined constraints were set up to represent average SUPERCODE systems runs as described above. For each of the three cases, the constraint interdependencies were varied such that the minimum evaluation scheme would execute between 1%-5% of the defined constraints per system change. Specifically, the three tests executed on average 1%, 3%, and 5% of the defined constraints per system change. Run-times are measured for both the naive checking scheme and the minimum evaluation scheme for the three different cases.

The minimum evaluation scheme only checks those constraints related to a single system change, but the number of constraints checked on average varies with each test. The minimum evaluation scheme executes on average 1.5 constraints per system change for the 1% case. For the 3% and 5% cases, the minimum evaluation scheme executes on average 4.5 and 7.5 constraints respectively per system change. The average run-times were gathered by timing the minimum evaluation scheme for each system change and averaging over the total number of system changes.

$$AverageMinimumEvaluationRuntime = \frac{\sum_{1}^{TotalSystemChanges} MinimumEvaluationRuntime}{TotalSystemChanges}$$

We expect the run-time for the minimum evaluation scheme to increase as the percentage of the executed constraints increases for two reasons: the number of constraints executed is greater, and the condition clauses are more complex. The more interdependent the condition clauses are, the more complex they are. Although the naive checking scheme will always check 150 constraints, the more complex error detection expressions of the higher percentage cases will increase its run-time as well.

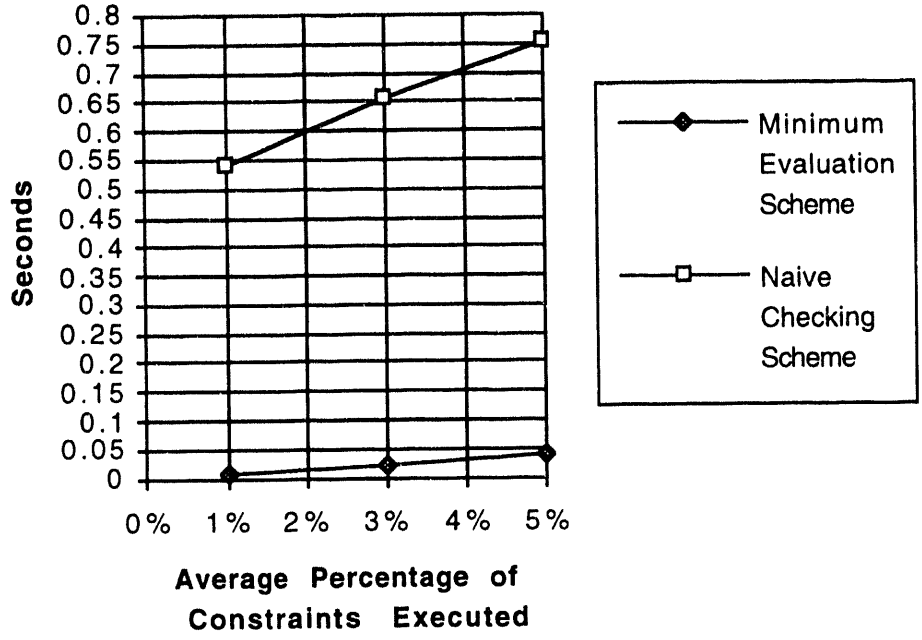

Figure 14: Average run-times per system change for average SUPERCODE *cases with 150 defined constraints: 1%, 3%, and 5% of constraints executed. Run-times for the naive checking scheme are also shown.*

Evaluation Figure 14 shows the run-times for both the minimum evaluation scheme and the naive checking scheme for the three average SUPERCODE cases. As expected, the run-time for the minimum evaluation scheme increases as the percentage of the constraints executed and the condition clause's complexity increases. The additional time required to execute the more complex error detection expressions is depicted in the increasing run-times of the naive checking scheme.

The run-time behavior of both checking schemes is fairly predictable for cases of varying sizes. The minimum evaluation scheme run-time will increase as the average percentage of the total constraints executed and the condition clause's complexity increases. The naive checking scheme run-time will increase as both the total number of defined constraints and the condition clause's complexity increases. To demonstrate this, we examined smaller cases with 30 and 50 defined constraints. For both numbers of constraints (30 and 50), three cases were run such that the minimum evaluation scheme executed on average 1%, 3%, and 5% of the defined constraints per system change. These run-times and the run-times for the naive checking scheme for each of the six cases are presented in Figures 15 and 16.

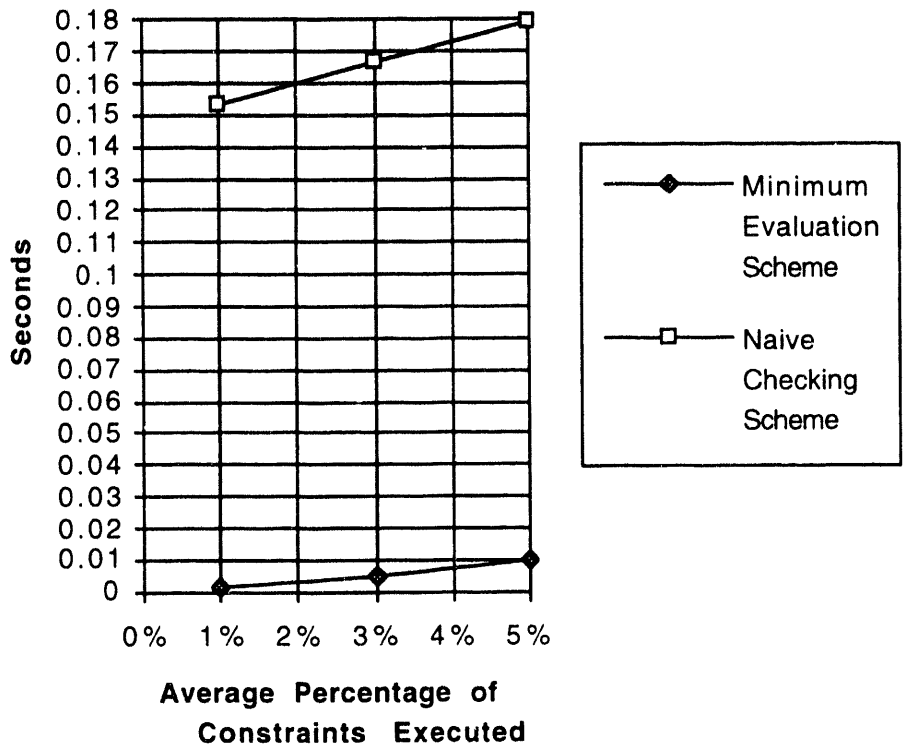

Figure 15: Average run-times per system change for average SUPERCODE *cases with 50 defined constraints: 1%, 3%, and 5% of constraints executed. Run-times for the naive checking scheme are also shown.*

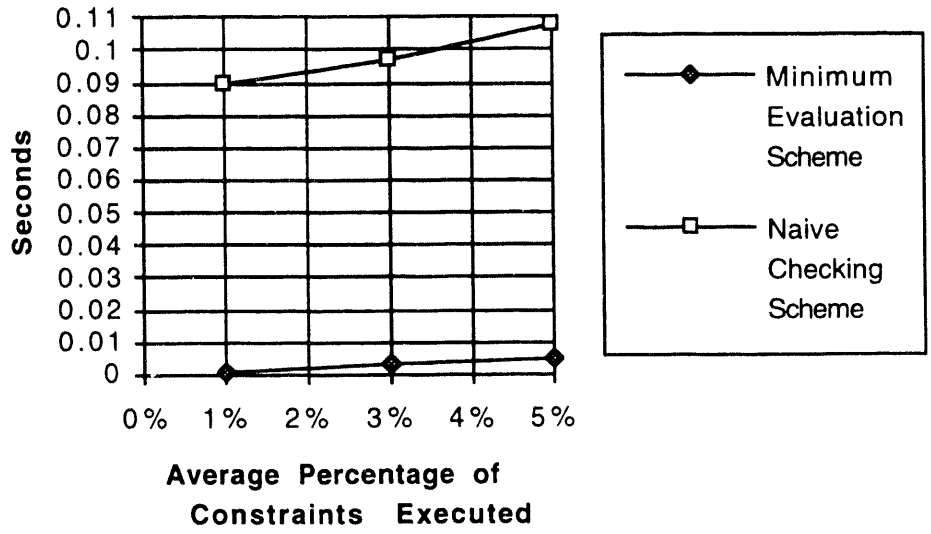

Figure 16: Average run-times per system change for average SUPERCODE *cases with 30 defined constraints: 1%, 3%, and 5% of constraints executed. Run-times for the naive checking scheme are also shown.*

As predicted, Figures 15 and 16 show that the run-time of the minimum evaluation scheme increases as the average percentage of the total constraints executed and the error detection expression's complexity increases. In addition to the longer run-times of the naive checking caused by the more complex error detection expressions, a comparison of the three different constraint sizes (30, 50, and 150 constraints defined) shows the significant run-time impact that the total number of constraints has on the naive checking scheme. This impact is much less for the minimum evaluation scheme.

Significance These cases show that the run-time nature of both checking schemes is predictable and that the minimum evaluation scheme significantly outperforms the naive checking scheme. Because both checking schemes are non chaotic, no further tests need to be conducted. The minimum evaluation scheme significantly outperforms the naive checking scheme for the average SUPERCODE cases and is, therefore, a success.

4.3 End-User System Success

The minimum evaluation scheme did significantly optimize the response time over the naive checking scheme, but is the response time fast enough to encourage use? Remember that the end-user response time includes these steps:

1. Sending a system changing command from the front-end to the shell.
2. Processing the commands.
3. Checking the constraints.
4. Sending response from the shell to the front-end.
5. Processing the shell's response.

The minimum evaluation scheme reduces the response time by greatly reducing step (3), but steps (1), (2), (4), and (5) must be measured to obtain the total response time to which the casual users are subjected. Shneiderman [25] has suggested guideline response times for different tasks. Typing and cursor motion should range between 0.05 and 0.15 seconds, and simple frequent tasks should take less than one second. There is no guideline established for our particular task, but we assess the task of using a dynamically configured GUI to use SUPERCODE to fall somewhere between these tasks. Shneiderman also points out that empirical testing can help set suitable response times. Therefore, empirical testing is used to determine user group satisfaction and the success of the system.

4.3.1 Total Response time

The total response time of interest is the response time after an error has occurred. This user environment is unique in that it allows the user to continue entering data into the interface while the underlying application is processing. This processing includes checking the constraints and sending error messages to the front-end. If no errors have occurred, the error detection system is transparent to the user. Therefore, we are interested in the response time indicated by the previously listed five steps only when step (4) involves sending the front-end an error message and step (5) displays the error message. Additionally, since user interaction is interrupted when the first error message is displayed, this response time is only relevant up to displaying the first error message. If the time required to display any additional error messages was longer than it takes the user to deal with the first error message (i.e., several seconds), this time would be relevant, but fortunately, this is not the case.

We have already presented the time required to check the constraints [step (3)] for various situations. and now we present the times required by the other portions affecting the end-user response time. These times are obtained by actual testing and should be considered approximations.

[step (1)] The time required by the front-end to send a system changing command and a check command to the shell is 0.01294 seconds.

[step (2)] Processing the commands (by the shell) takes 0.02003 seconds. The time for step (2) includes parsing and executing the system-changing command, parsing the check command, and calling the function that checks the constraints. Including the function call time was necessary because this time cost was not included in the times gathered for step (3).

[step (4)] Executing a typical action routine in the shell that sends an error message to the front-end takes 0.00604 seconds.

[step (5)] Finally, the time it takes the GUI to receive and display a typical error message is 0.02036.

Therefore, the total cycle time not including checking the constraints is 0.05937. The run-times for the minimum evaluation scheme for the average SUPERCODE cases (150 constraints defined) with the rest of the cycle time added is depicted in Figure 17.

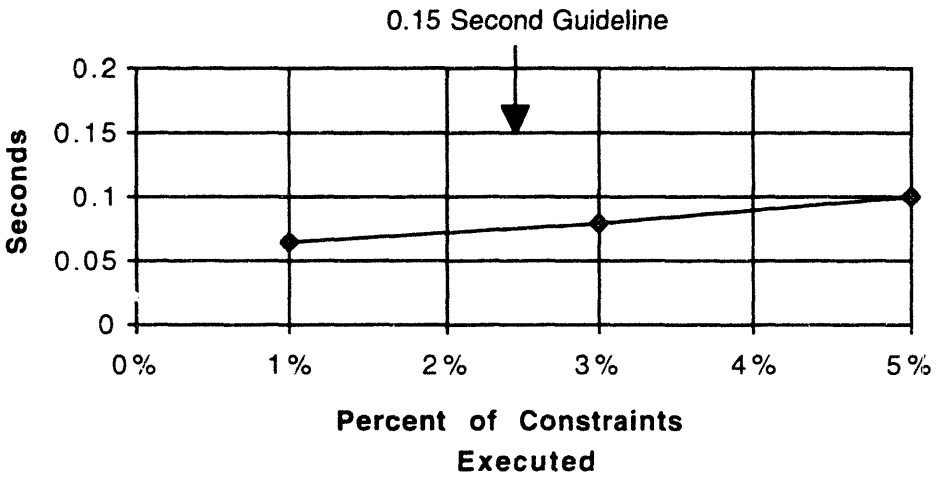

Figure 17: Minimum evaluation scheme run-times for the average SUPERCODE *case including cycle time with 150 constraints defined. The 0.15 second guideline is marked with an arrow.*

Recall that Shneiderman suggested a guideline response times between 0.05 and 0.15 seconds for tasks like typing, cursor motion, and mouse selection. We consider using SUPERCODE with a GUI to be a more difficult task than simple typing or cursor motion, but notice when the cycle time is added to average SUPERCODE cases, the minimum evaluation scheme run-time still stays below 0.15 seconds. The maximum run-time presented for the minimum evaluation scheme is approximately 0.1 seconds. These results are very promising, but to determine whether the response time is indeed low enough, user testing was used.

4.3.2 User Feedback

The next step is to present the system to the end-user group and to question them as to what they liked and disliked. We are interested in overall satisfaction, whether or not they would use it, etc. The users were given a GUI specification file to read into the shell and a set of tasks to perform. After completing the test, the users filled out a questionnaire that addressed usability and asked for criticisms, comments and, suggestions. There were some criticisms and suggestions, but overall, the end-user environment was a solid success.

The users found using the GUI very easy and preferred it to using the command-line version. Some of their comments are presented:

> The fact that one doesn't have to remember all the variable and constraint names is very important. From my experience, I find that if I don't run the SUPERCODE for a while, I have a hard time remembering the names of variables etc., and I have to look at the *.mod files. Most of us also run several other transport/MHD codes with different names for the same physical quantities, and this makes it even more confusing.
>
> I liked it because it makes it much easier to run the SUPERCODE without having to know much about the shell. The different choices of parameters in the different menus and submenus were well chosen, and I believe that anybody with a knowledge of the MHD would be able to run a case.
>
> It makes tasks like this easy to do, and easy to learn from. Plus I can still go up to the shell [terminal window] and do arbitrary programming tasks. For example, now that I've seen what happens when I run an equilibrium, I could try something more complex like setting up a for-loop to run equilibria for a range of values of one of the variables.
>
> It's much easier than activating/deactivating constraints, variables, and calculators by hand. I never remember the names of all those thing and constantly have to search for them.
>
> When I haven't used SUPERCODE for a month or so, I forget the variable names. Sometimes it can take me MINUTES to find the correct one (there are > 100 to choose from). Accordingly a GUI is an ESSENTIAL feature in my opinion.

When asked whether or not the users felt the system would promote distribution to users who don't know the implementation specifics of SUPERCODE, all of them said yes and cited the reasons why they themselves liked it. One user added, "This is probably the biggest use for the GUI."

The users offered very useful comments and suggestions. The front-end and SUPERCODE are two separate applications, and to use the system, the user currently has to open two x-terminals and start each application separately. One user felt that launching SUPERCODE from within the front-end would be a good refinement. Also, the GUI displays both warning and error messages. It was suggested that if an error message is displayed on a user input, displaying a warning message that is also defined for that input is superfluous and should not be displayed. Other suggestions for enhancements were an on-line help system, providing a menu of standard graphs that could be automatically displayed when the equilibrium were recalculated, and enhancing the shell and the front-end to enable them to display pertinent calculation feedback (i.e., the calculation failed because . . .). Because the users found the end-user environment very usable and felt it would promote distribution, the end-user environment is a success.

Chapter 5

Conclusions and Future Work

The prototype has returned encouraging results. In this section, we summarize the issues, discuss the success of the prototype, and indicate topics for future work.

5.1 Conclusions

To help promote distribution of codes, it is desirable to provide a friendly end-user environment. Accordingly, we designed a GUI creation system to allow construction of easy-to-use GUI's and a constraint system to provide error detection and handling. We implemented a prototype of this system for SUPERCODE, a tokamak design code.

The end-user community's comments in their surveys indicate that the system made the SUPERCODE easier to use. The constraint checking system ensured that their answers did not violate the appropriate system model, and the users especially liked the robust quality of the system. The end-users also found the interface fast enough for the kinds of problems they address. From all this, we can infer that the prototype's interface style can promote the accessibility and usability of scientific codes.

GUI designers found the GUI creation system easy to use and felt that any initial extra effort involved in creating GUI's was worth the benefit of increased code distribution. We believe that the system is powerful enough to be applicable to other scientific codes.

5.2 Future Work

A few modifications presented themselves that could enhance usability. One possibility is building a smarter preprocessor or adding to the shell language to support the constraint system instead of using the current preprocessor. This allows the system to handle read sets over function calls. Other enhancements include adding a generic interface to the front-end and responding to user suggestions. These are all areas for further work.

5.2.1 Alternate Implementations to the Current Preprocessor

Building a smarter preprocessor using a parser generator like YACC [26] or modifying the shell language to support the constraint system are improvements over the current preprocessor. Both implementations would improve syntax and would be able to properly deal with function calls within condition clauses. However, the smarter preprocessor could be used to read code that is to be compiled while the modified shell version could not. On the other hand, the modified shell version could automatically generate and use reference variables for static class variables while the smarter preprocessor could not. These tradeoffs should be seriously considered before choosing an alternate implementation to the current preprocessor.

5.2.2 Generic Front-end Enhancement

Once the constraint system is implemented in the shell language, it is possible to construct a generic GUI for the front-end by accessing system information. Because the shell can access all system information, the front-end can get this information by querying the shell. The GUI can query the shell to determine how many variables there are and their associated data (type, current value, etc.). The front-end could then create a window to display this information. In addition, a GUI can be made so that the user can select a variable from this window for modification and/or definition of a constraint.

This generic GUI could be expanded to include access to SUPERCODE's equation, set includes figures of merit, physical constraints, and calculators. A generic GUI that allows access and manipulation of the system equation set would be extremely beneficial.

5.2.3 User Feedback Suggestions

There were several suggestions made by the users that are considered topics for future work: enabling the user to launch SUPERCODE from inside the front-end, modifying the warning and error message display system so that warning messages are not displayed for user inputs that also generated an error message, adding an on-line help system, providing a menu of standard graphs that could be automatically displayed when equilibrium was recalculated, and enhancing the shell and the front-end to enable a display of pertinent calculation feedback.

Bibliography

[1] S.W. Haney, W.L. Barr, J.A. Crotinger, L.J. Perkins, C.J. Solomon, E.A. Chanoitakis, J.P. Freidberg, J. Wei, J.C. Galambos, J. Mandrekas, "A 'SUPERCODE' for Systems Analysis of Tokamak Experiments and Reactors," *Fusion Technol.*, **21**, pp. 1749-1759 (1992).

[2] P.F. Dubois, Z.C. Motteler, P.A. Willmann, R.A. Allsman, C.M. Benedetti, S.M. Hockett, D.S. Kershaw, A.B. Langdon, A.C. Springer, J. Takemoto, S.S. Wilson, "The Basis System," Manual M-225, Lawrence Livermore National Laboratory, Livermore, CA (1989).

[3] L.D. Pearlstein, J.A. Crotinger, S.W. Haney, L.L. Lodestro, "CORSICA 1.0: A Free-Boundary 1-1/2 D Transport Code," *Bulletin of the APS*, **38:10**, pp. 2077 (1993).

[4] A. Benson and G. Aitken, *OI Programmer's Guide*, Prentice Hall, Englewood Cliffs, NJ (1992).

[5] *agX/ Toolmaster User's Guide*, UNIRAS A/S, 2nd ed., Denmark (1991).

[6] Sun Microsystems, Inc., *OpenWindows Developer's Guide 1.1 User's Manual*, Sun Microsystems, Inc., USA (1990).

[7] M.K. Mahoney, *Tutorial Notes from the ACM Conference on Human Factors in Computing Systems*, Monterey, CA (1992).

[8] B.A. Meyers, D.A. Giuse, R.B. Dannenberg, B.V. Zanden, D.S. Kosbie, E. Pervin, A. Mickish, P. Marchal, "Garnet: Comprehensive Support for Graphical, Highly Interactive user Interfaces," *Computer*, **23**, pp. 71-85 (1990).

[9] C. Upson, T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, A. van Dam, "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE Computer Graphics & Applications*, **9:4**, pp. 30-42 (1989).

[10] J. B. Goodenough, *CACM*, **18:12**, pp. 683-696 (1975).

[11] J.M. Noble, "The Control of Exceptional Conditions in PL/I Object Programs," *Proceedings of the IFIP Congress 68*, North-Holland Publishing Co., Amsterdam, pp. C78-C83 (1968).

[12] B. Liskov and A. Snyder, "Structured Exception Handling," Computation Structures Group Memo 155, Massachusetts Institute of Technology, Cambridge, MA (1977).

[13] *Reference Manual for the ADA Programming Language*. United States Department of Defense, ANSI/MIL-STD-1815A-1983, (1983).

[14] J.G. Mitchell, W. Maybury, and R. Sweet, "MESA Language Manual," Xerox Research Center, Palo Alto, CA (1979).

[15] S. Yemini, "The Replacement Model for Modular, Verifiable Exception Handling," Ph.D thesis, University of California, Los Angeles, CA (1981).

[16] C. Dony, "An Exception Handling System for an Object-Oriented Language, *Proceedings of ECOOP'88*, pp. 146-161 (1988).

[17] M.A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA (1991).

[18] A. Goldberg and D. Robson, *SMALLTALK 80, the Language and its Implementation*, Addison-Wesley, Reading, MA (1983).

[19] B. Meyer, *Eiffel The Language*, Prentice Hall, London, England (1992).

[20] C. Dony, "Exception Handling and Object-Oriented Programming: towards a synthesis,"*Proceedings of ECOOP/OOPSLA '90*, 322-330 (1990).

[21] R. Levin, "Program Structures for Exceptional Condition Handling," Ph.D thesis, Carnegie Mellon University, Pittsburgh, PA (1977).

[22] M.E. Lesk and E. Schmidt, "Lex: A Lexical Analyzer Generator," in B.W. Kernighan and M.D. McIlroy, *Unix Programmer's Manual*, 7th ed., Bell Laboratories (1978).

[23] A. I. Holub, *Compiler Design in C*, Prentice Hall, NJ (1990).

[24] M. Marcotty and H. Ledgard, *Programming Language Landscape: Syntax, Semantics, and Implementation*, 2nd ed., Macmillan, New York, NY (1986).

[25] B. Shneiderman, *Designing the User Interface Strategies for Effective Human-Computer Interaction*, 2nd ed., Addison-Wesley, Reading, MA (1992).

[26] S. C. Johnson, "Yacc: Yet Another Compiler-Compiler," in B. W. Kernighan and M. D. McIlroy, *Unix Programmer's Manual*, 7th ed., Bell Laboratories (1978).

DATE FILMED

4 / 19 / 94

END