A User-Extensible Architecture for Visualization and Analysis of Time-Series Trace Data

by

Alan L. Batongbacal

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

Prof. Marc Abrams, Chairman

Prof. Dennis Kafura

Prof. Sallie Henry

April, 1996

Blacksburg, Virginia

A User-Extensible Architecture for Visualization and Analysis of Time-Series Trace Data

by

Alan L. Batongbacal

Committee Chairman: Prof. Marc Abrams

Computer Science

(ABSTRACT)

This thesis describes the design and implementation of CHITRA95, a software system developed for the visualization and analysis of time-series trace data. CHITRA95 is based upon two earlier generations of CHITRA and is aimed at producing a system with broad applicability and utility in this area of research.

This thesis contributes to the area of software design for trace visualization and analysis by proposing a set of design principles towards achieving the goals of system extensibility, reusability, reliability, testability and verifiability.

These design principles are demonstrated by CHITRA95, a software architecture proposed in this thesis for visualization and analysis of time-series trace data. This architecture is novel in its combination of independence from problem domain semantics; optimization for user-extensibility and code reusability; freedom from any specific user interface model; ability to simultaneously produce an integrated application and a reusable toolkit of parts that may either be customized into a turnkey system or integrated into other software systems; support for enhanced reliability, testability and verifiability; and support for an interface to the World Wide Web and for remote execution. Finally, this thesis makes the specific contribution of a data structure for representing large traces that permits the maintainance of multiple versions of a trace and retains the ability to undo modifications made to a trace.

ACKNOWLEDGEMENTS

I wish to express my gratitude to the following people without whom this thesis would not have come to fruition: my parents Atty. Mario and Rosario Batongbacal, sisters Rosa Carmina and Donna, and brother Jay, for their love and unwavering support; my advisor, Dr. Marc Abrams, for his guidance, patience and understanding; my committee members, Dr. Sallie Henry and Dr. Dennis Kafura; all the members of the Siochi family, in particular Mrs. Loiva Siochi and Mr. Andres Siochi, for their prayers and fellowship; and my friends in the Filipino Student Association and the Council, whose gentle cajoling pushed me along when I needed it the most.

I wish to thank John Paul Vergara and Fernando Siochi in particular, whose counsel and guidance carried me through frequent rough spots in my life, and helped me attain a better understanding of myself and others. I especially appreciate all those long psychoanalysis sessions long into the night; they have been most enlightening.

Finally, I wish to thank Pats, Elle and Collette, for collectively teaching me the finer points of love, friendship and trust.

TABLE OF CONTENTS

1 INTRODUCTION **1**

1.1 Motivation 3

1.2 Problem Statement 4

1.3 Contributions of This Work 5

1.4 Solution Overview 6

1.5 Related Work 9

1.5.1 The UNIX Operating System 9

1.5.2 *Mathematica* 10

1.5.3 Pablo 10

1.5.4 Traceview 11

1.5.5 Chitra93 12

1.6 Summary 13

1.7 Thesis Overview 13

2 A BRIEF TOUR OF Chitra95 **14**

2.1 The Chitra95 Command Set 14

2.2 How Chitra95 Modules Communicate 15

2.3 A Sample Trace 18

2.4 Ensemble Creation 19

2.5 Ensemble Output 19

2.6 Model Generation 22

2.7 Trace Synthesis and Model Validation 23

2.8 Version Control 25

2.9 A Note About Chitra95's Canonical Format 26

3 DESIGN PRINCIPLES **31**

3.1 Use the toolkit approach to software construction. 31

3.1.1 Propagate object orientation throughout the design. 32

3.1.2 Factor out common operations. 32

3.1.3 Prepare to be assimilated. 32

3.2 Separate data visualization from data generation. 33

3.2.1 Expect new viewers to be added. 33

3.3 Automate user interface construction. 33

3.4 Save time and effort with non-interactive batch testing. 34

4 THE DESIGN OF Chitra95 **35**

4.1 Logical Structure 35

4.1.1 Terminology and Notational Conventions 37

4.2 The Trace Representation 38

4.3 The Ensemble Representation 43

4.3.1 Ensemble Construction 43

4.3.2 Ensemble-level Queries and Statistics Computation 45

4.3.3 Translation Between Trace Records and Trace Record Encodings . . 46

4.3.4 Version Control 46

4.4 The Inserter/Extractor Layer 47

4.5 The Generator/Transformer/Synthesizer Layer 49

4.5.1 Generators 49

4.5.2 Transformers 52

4.5.3 Synthesizers 53

4.6 The Presenter Layer 54

4.7 The User Interface Layer 56

4.8 Module Verification 58

4.9 Summary 60

5 THE IMPLEMENTATION OF Chitra95 61

5.1 Implementation Choices 61

5.2 Organization 62

5.3 Building Blocks 63

5.3.1 Basic Data Types 63

5.3.2 Abstract Data Types 63

5.3.3 Base Classes 66

5.3.4 The `CFStream` Class 67

5.3.5 Other Classes and Functions 69

5.4 The Ensemble Representation 71

5.4.1 The `Member` Class 71

5.4.2 The `StateMap` Class 74

5.4.3 The `Ensemble` Class 75

5.5 The Inserter/Extractor Layer 76

5.6 The Generator/Transformer/Synthesizer Layer 78

5.7 The Presenter Layer 78

5.8 The Interface Layer 79

5.9 Module Verification 80

5.10 Summary 80

6 LESSONS LEARNED 82

6.1 A modular architecture is flexible but slow. 82

6.1.1 The cost of modularity 83

6.2 New interface models interact with design decisions. 84

6.3 Literate programming is wonderful if employed properly. 84

6.4 Anticipate shifts in tool usage. 85

6.4.1 An embedded database manager can be useful. 86

6.4.2 Even subtle shifts in emphasis can have important repercussions. . . 86
6.5 There is always room for design refinement. 87

7 CONCLUSIONS **88**
7.1 Thesis Summary 88
7.2 Future Directions 89

A THE Chitra95 KERNEL API **93**
A.1 Opening and closing an ensemble 93
A.2 Adding a new member 94
A.3 Reading member data 94
A.4 Appending new records to a member 95
A.5 Starting a new version 96
A.6 Editing member data 96
A.7 Undoing changes to the ensemble 97
A.8 Computing ensemble and ensemble member statistics 97
A.9 Other ensemble operations 98
A.10 Input/output: `CFStream` programming concepts 98
A.11 Reading and writing attributes 99
A.12 Reading and writing data tuples 101

LIST OF FIGURES

2.1 The CHITRA95 Command Set 16
2.2 A Sample Trace File 18
2.3 Output of `e.all` with absolute timestamps 20
2.4 Output of `e.all` with occupancy times 21
2.5 Output of `v.text` 22
2.6 Graph created by `v.gantt` 23
2.7 Output of `m.sm.ens` for ensemble `demo.ens` 24
2.8 Output of `v.matrix` for ensemble `demo2.ens` 25
2.9 Output of `x.a.fil.ens` 26
2.10 The ensemble after application of the filter aggregation transform 27

4.1 Logical Structure of CHITRA95 35
4.2 CHITRA95 Module Interaction Model 58
4.3 *WebScope* Prototype User Interface 59

5.1 General structure of a `CFStream` 67
5.2 A freshly-created ensemble member 72
5.3 The ensemble member after inserting a new record at position 0 73
5.4 The ensemble member after deleting the record at position k 73

LIST OF TABLES

2.1 Some Commonly-Used CHITRA95 Modules 17
2.2 Common Global Attributes 29
2.3 Common Object-Specific Attributes 30

4.1 Primitive Trace Representation Operations 39
4.2 Ensemble Member Operations 41
4.3 Ensemble Operations 44

6.1 Comparative timings (in seconds) for CHITRA93 and CHITRA95 83

Chapter 1

INTRODUCTION

Consider the following scenarios. A World Wide Web server administrator looks for patterns that characterize accesses to the server before making an attempt to optimize the organization of the files that the server holds. A government planner pores through mountains of census data in search of geographical trends that will influence resource requirements for the next decade. A police psychologist dissects the personal records of a serial killer's victims in the hope of identifying shared traits that may give the clue to his identity. These problems and many others like them require analysis of trace data, a trace being a collection of observations of one or more features of interest in some entity or system over some specific period of time.

This thesis presents a software architecture that supports the development of tools and techniques for the analysis of traces. This architecture is based upon a set of design principles that may be applied to any software for trace visualization and analysis. The architecture is implemented by the topic of this thesis, CHITRA95, an integrated application for visualization, modeling and statistical analysis of trace data. At the same time, CHITRA95 can also be described as a toolkit for building larger, possibly more specialized applications in the future for trace analysis; this feature is of particular importance to researchers faced with new problem domains of trace data which often require new, unforeseen methods of trace analysis. The product of five years of on-going research, CHITRA95 builds upon the lessons learned from two preceding generations of tools for trace analysis [4, 5].

A trace represents the behavior or evolution of some entity or system which we call the *target*. Given any arbitrary target, one or more features of which are recorded in a trace, it is often the case that the trace may contain sequences that vary considerably with

respect to the conditions under which the trace was collected. For example, a trace that lists the size of packets retrieved from a network by a network sniffer may reflect variations that correspond to user loads that change over the course of a day. Thus a representative characterization of the target may require simultaneous analysis of more than one trace in order to capture variance with respect to collection conditions.

CHITRA95 is designed to work with one or more traces at a time. A set of traces analyzed as a unit is called an *ensemble*; each trace in an ensemble is referred to as a *member*. Each member of an ensemble represents a specific instance of the target and is typically characterized by one or more target inputs. For example, the network sniffer trace described previously may be characterized by the number of users present on the network or the type of protocol used by the networking software. Indeed, an ensemble may be composed of traces which differ in one or more of these inputs, although the traces may simply be taken from instances given the same inputs. CHITRA95 also provides facilities for identifying and working with subsets of the ensemble (i.e., *sub-ensembles*).

One important problem tackled by CHITRA95 is how an ensemble can be analyzed without exhaustive visualization. For example, in the course of a recent study [6] of file sizes and types accessed during World Wide Web "surfing" sessions of two different populations of students, outliers were discovered only after exhaustive visualization of lengthy server traces. Compounding the problem is the fact that many ensembles consist of a large number of arbitrarily large traces (e.g., a total of 2 gigabytes of traces have been collected to date in connection with the aforementioned study), leading to the *scale problem* in trace analysis: how can an analysis tool both simultaneously analyze a large number of traces and present its results without subjecting the user to cognitive overload.

CHITRA95 solves this problem by providing novel methods of sifting through large amounts of data, much of which may be irrelevant; these methods fall into the categories of *data reduction* and *data partitioning*. Data reduction consists of reducing the amount of irrelevant data present in a trace by generating a representation of the trace's essential features. Data partitioning consists of identifying those members of an ensemble which

comprise groups; each group can then be manipulated individually. CHITRA95 provides data reduction by means of *transforms* and *models*. A transform reduces the length of a trace either by physically deleting one or more portions or by collapsing some subsequence according to some rule. On the other hand, a CHITRA95 model is an estimate of the stochastic process underlying a trace or ensemble. With this approach, the problem of understanding the observed behavior is reduced to that of finding a compact, presumably easier to understand representation of the original trace. Data partitioning in CHITRA95 is achieved with the aid of statistical tests of homogeneity.

No tool for trace analysis can anticipate every problem domain or accommodate newly devised trace analysis methods, and CHITRA95 is no exception. Nevertheless, by allowing for incremental refinement with low attendant programming costs, a tool may increase its usefulness to and retain the interest and loyalty of its audience. The software architecture presented in this thesis and implemented by CHITRA95 emphasizes the ability to accommodate user extensions and enhancements. The usefulness of a tool is not merely a function of its capabilities—we contend that it is just as important for the tool to be able to accept augmentation and ultimately adapt to the evolving needs of its users.

1.1 Motivation

In order to answer the question "Why build CHITRA95?," it is necessary to briefly delve into the history of the earlier versions. The first two generations of CHITRA, CHITRA91 [8] and CHITRA93 [1], were monolithic programs with tightly integrated graphical user interfaces (GUIs). The complexity and level of integration were such that it was difficult to make major modifications or enhancements. Problems related to code duplication, side effects and undesirable interactions crept into the tool; the tool became larger with each new feature and consequently became increasingly difficult to debug, let alone verify. With respect to CHITRA93 in particular which was the focus of much programming effort by several programmers, it was evident that improvements were needed, particularly with

regard to the tool's reliability and maintainability. There was also the desire to broaden CHITRA's appeal by making it available on other platforms; CHITRA's portability became an issue.

At the same time that difficulties in improving the tool were being encountered, basic assumptions concerning the range of problems tackled by CHITRA, its audience and its primary objective were constantly challenged and subsequently revised. CHITRA was originally conceived in part as a tool for generating performance models of traces of numeric data; with the advent of traces of non-numeric data (i.e. categorical data) with either absolute time stamps or no time stamps at all, not only were new techniques to handle the data needed, but the design of the tool had to be recast as well. CHITRA was initially aimed at creating models for performance analysts; its expanded capabilities extended its appeal to a different class of users, one not necessarily concerned with performance models and often not primarily interested in programming. Indeed, CHITRA's very objective shifted from being a tool for generating models to being a tool for exploring data for various objectives. It became clear that the logical solution was to evolve towards a modular architecture, primarily to ease extension and enhancement. The design of such an architecture is the focus of this writer's research; CHITRA95 is the reference implementation of that architecture.

It should be noted that almost all other trace analysis tools are limited in some degree in their ability to accommodate user extensions and enhancements. Thus, the software architecture presented in this thesis is of relevance not only to the CHITRA research effort but also to the field of trace analysis tool development in general.

1.2 Problem Statement

The problem dealt with by this thesis may be stated thus: *Define and implement an architecture for a software system for the visualization and statistical analysis of trace data that emphasizes extensibility, reusability, portability, reliability and maintainability and accommodates user augmentation and enhancement without requiring an unreasonable amount*

of system-specific knowledge or programming skill.

1.3 Contributions of This Work

This thesis presents a software architecture for trace analysis and visualization based upon a set of design principles that are applicable to the design of any trace analysis tool. The architecture is novel in the following ways:

- The architecture is problem domain independent to the extent of allowing the user interface to be tailored to the problem domain. No problem domain semantics are embedded into the architecture, which permits the analysis of any type of trace file.

- The architecture is highly modular and is optimized for user extensibility. The use of a text-based canonical file format for data exchange among existing modules permits users to build new modules using the programming or scripting language of their choice; no linking is required.

- The architecture promotes module reusability by stressing the creation of numerous, well-defined modules.

- The architecture is portable, both in the traditional sense of being machine and operating-system independent and, more importantly, in its independence from any specific user interface model. By separating data computation from data display, it limits the amount of domain knowledge required of the module writer to that strictly relevant to the computation desired. By automatically generating a user interface, it promotes consistency and usability without disproportionately adding to the programming burden. By stressing content over appearance, it allows for an interface that is adapted to the audience's needs and appropriate to its level of skill. This degree of portability, combined with automatic user interface generation and the architecture's emphasis on extensibility makes it possible for a non-dedicated programmer such as a

specialist in statistics, performance analysis or visualization to rapidly and seamlessly add new modules.

- The architecture is flexible, in the sense that it supports the creation of a cohesive, integrated tool for trace analysis while simultaneously providing a toolkit of parts that can be selected from either to form a customized tool for specific types of analysis or to be integrated into another trace analysis tool from a third party.

- The architecture fosters reliability, testability and verifiability. Modularity improves reliability by localizing failures such as memory errors which are typically limited to one module. By factoring out common code, source code size is reduced, and maintainability is enhanced. At the same time, modularity promotes the creation of small, easily written and verified test suites and the ability to diagnose which module is at fault whenever errors are encountered.

- The architecture allows for the addition of a World Wide Web interface and permits remote execution. By requiring modules to use standard input and standard output and to be executable via command-line invocation, the architecture allows a conforming tool to be used with a scheme such as the Common Gateway Interface (CGI) [10].

In addition, this thesis contributes to the field of software design for trace analysis with the introduction of a data structure that supports the maintainance of multiple versions of traces and permits the arbitrary modification of large traces with the ability to undo the modification at any time.

1.4 Solution Overview

CHITRA95 is organized into logical module groups, each of which encompasses a number of small modules with well-defined functions. The modules are organized into layers, each of which builds upon the one underneath it. More specifically, the bottom layer implements an

interface to an ensemble representation. The next layer consists of modules that facilitate trace data insertion and extraction. Following that is a layer consisting of analysis and transformation modules. The results that come from this layer are then passed on to the next layer consisting of modules for presentation in various fashions such as text dumps and graphs. Finally, the top-most layer consists of modules that implement user interfaces; these modules give CHITRA95 a unified, coherent appearance that facilitates ease of use. By enforcing the use of standard input and standard output along with a canonical file format as the means of communication between modules, the CHITRA95 user is able to take advantage of pipelining and make full use of the set of text-processing filters available on the host operating system (e.g., the *awk*, *sed* and *grep* programs available on all Unix systems). Also, by maintaining separation between logically distinct modules, the CHITRA95 user can easily add new functionality simply by writing a new module in the language of her choice, provided that the new module communicates via standard input and standard output and makes use of CHITRA95's canonical file format.

Organizing CHITRA95 into logical module groups permits new modules to be developed in parallel by programmers working in large research groups. It also simplifies the task of exhaustively verifying each module's function through unit testing, something which is critical to promoting overall confidence in the tool. Moreover, modularization localizes failures, and the failures that do occur cannot easily crash unrelated code.

Last but not least, one hallmark of the CHITRA95 design is the amount of attention paid to improving portability to other hardware platforms, operating systems and/or GUI toolkits.

CHITRA95's approach to the first two, platform and operating system portability, is conventional—the developers have taken care to eliminate as many assumptions as possible about the underlying hardware (e.g. whether the architecture is big-endian or little-endian, whether the word length is 32 bits or 64 bits) and have written all code either in ANSI C or in a non-controversial subset of C++ . The software architecture upon which CHITRA95 is based includes special measures to allow the tool to remain independent of any specific

GUI toolkit. This is not to say that CHITRA95 is designed to be used solely from the command-line (an approach that would immediately rule out the Apple Macintosh as a potential platform). On the contrary, CHITRA95 uses a high-level, input/output-oriented approach to specifying how its modules interact with the user. This is achieved by providing each module with a configuration file that specifies in general terms what input the module takes and what output it generates. The command-line syntax, for example, is derived from the configuration file. The same configuration file is also used to generate a graphical user interface with the assistance of a GUI toolkit-specific assistant. The generic configuration file approach's greatest strength lies in its ability to generate interfaces for multiple GUI toolkits.

Tool implementation, like all other software development, is centered on writing, debugging, testing, validating and maintaining source code; but it must be done in the context of a large group whose members possess disparate levels of programming and documentation skill. CHITRA95 adopts a toolkit approach to building software: instead of being a monolithic, one-use tool, it is built from a set of independent programs that together form a coherent, multi-featured tool. At the same time, this set of independent programs provides a toolkit of modules that can be used independently and even integrated into other tools, such as a computer-aided software engineering (CASE) tool. One powerful advantage gained by CHITRA95's modular architecture is the ability to easily identify and extract a custom subset of the independent programs tailored to a specific task. This ability is made possible because the programs' respective configuration files can be easily modified to use appropriate language and interface organization. Such a *turnkey system* can be rapidly deployed, if necessary; at the same time, customization can be achieved quickly. An example of a CHITRA95-based turnkey system is one called "WebScope" developed and used by the SUCCEED research group at Virginia Tech to analyze World Wide Web server traces [6]. Another benefit of CHITRA95's modular architecture is the ability to reuse individual modules as part of other software systems by adding glue code to translate to and from CHITRA95's canonical file format. For example, the Pablo System [17] contains a rich set

of tools that can integrate with CHITRA95 simply by adding a filter to translate between Pablo's SDDF (Self-Defining Data Format) and CHITRA95's canonical file format.

1.5 Related Work

The design of CHITRA95 draws from a diverse set of influences, ranging from the UNIX operating system to end-user applications (e.g., *Mathematica*) to an assortment of tools developed specifically for software performance visualization and evaluation (the very area of interest that spawned the CHITRA project), the last including Pablo [17], Traceview [14] and CHITRA93 [1][1]. In this section, we briefly describe each of these and highlight the features that we incorporated into CHITRA95's design.

1.5.1 The UNIX Operating System

By far the greatest influence on CHITRA95, the UNIX philosophy was articulated by Doug McIlroy and quoted in [20] as follows:

> Write programs that do one thing and do it well. Write programs that work together. Write programs that handle text streams, because that is a universal interface.

The decision to break the design of CHITRA95 into small modules derived directly from this philosophy, as did the decision to use text files for CHITRA95's canonical file format. The use of text files also permitted CHITRA95 modules to be constructed from utilities already provided by UNIX (e.g. *sed* and *grep*), as well as from more powerful text-manipulation tools such as *awk* and, more recently, *perl*.

[1] In contrast with these three special-purpose tools, CHITRA95 is a general purpose trace analysis tool that is also usable for performance analysis.

1.5.2 *Mathematica*

Mathematica [21] is a powerful software application produced by Wolfram Research, Inc. Briefly stated, it is a system for performing numerical, symbolic and graphical computations. In addition to providing an exhaustive array of built-in functions, it also provides a means for creating interactive interfaces (called *notebooks*) for specific applications (e.g. tutorials and demonstrations) and permits its users to add functionality through custom-written *packages* of algorithms that "teach" *Mathematica* about particular application areas. CHITRA95 improves upon *Mathematica* by allowing enhancements to be written in the user's language of choice, as opposed to forcing the user to learn and use *Mathematica*'s notation and language. Nevertheless, the ability to create customized variations of CHITRA95 is a direct result of CHITRA95's developers' efforts to emulate the degree to which *Mathematica* can be enhanced.

1.5.3 Pablo

The Pablo Visualization Environment [17], or Pablo for short, is a system under development by Reed *et al.* at the University of Illinois at Urbana-Champaign. Pablo is described by its authors as a "toolkit for the construction of performance analysis environments." What this means is that Pablo, as currently implemented, is a collection of modules that fall into two categories: software instrumentation and data analysis. Communication between modules is facilitated through the use of a trace file meta-format.

The software instrumentation portion of Pablo is composed of a graphical interface with which the user may specify points in source code to be instrumented; C and FORTRAN parsers that generate modified versions of the source code that include calls to a trace capture library, and a trace capture library that lets the user manipulate and store trace files in the Pablo meta-format.

The data analysis portion is composed of data transformation modules that the user connects in a directed graph using another graphical interface. A trace file to be analyzed is

then fed into a module designated as input; other modules selected user-specified fields, while more modules further downstream may perform calculations upon their values. Through judicious module arrangement and configuration, the user can then compute the desired performance metrics.

In addition to tools for visualizing trace data, Pablo adds a sonification module that lets users hear a mapping of their data to sound with the aid of additional hardware. The authors report that sonification adds an important dimension to the analysis process, one which often reveals features not otherwise exposed by a visual display.

Pablo was designed with three major goals in mind. The first one, portability, is achieved by an approach that eschews architecture-dependent features, promoted through the use of the aforementioned trace file meta-format that contains no embedded semantics. The second goal, scalability, is attained partially with a message-passing architecture that lets Pablo's various modules be distributed among multiple processors. The third goal, extensibility, is achieved through well-defined modularization and is aided again by the use of a trace file meta-format. Although Pablo pursues its goals with considerable success, it fails to address the issue of reliability by not providing a clear method of verifying the proper function of individual modules, let alone user-defined modules. CHITRA95 owes in part to Pablo the notion of a canonical file format and the toolkit approach to modular decomposition along data transformation boundaries.

1.5.4 Traceview

Traceview, by Malony *et al.*[14], is a general-purpose trace visualization tool which in principle is closely related to the CHITRA system. Traceview takes any time-ordered trace and allows the user to (a) apply a selection function to generate a new, *virtual* trace; and (b) use a GUI to display the virtual trace in a number of ways, two of which are provided by Traceview itself (Gantt charts and rate displays). Traceview was designed to be extensible, in the sense that the user can add her own display methods. However, the user cannot display combinations of data from multiple traces, nor can she add her own data selection

functions (the only ones defined are trace clipping and event filtering). CHITRA95 reveals its Traceview heritage in its support for extensions through user programming and improves upon Traceview by placing no restrictions on the type and function of user-added modules other than that the new modules communicate their information in CHITRA95 canonical file format via standard input and standard output.

1.5.5 CHITRA93

CHITRA93 [1] is a software system developed at the Computer Science Department of Virginia Tech and is CHITRA95's direct ancestor. It provides the user with a GUI based on X11 and OSF/Motif. It operates on a *program execution sequence* (PES), which is a set of tuples representing the state of a program in execution. Program states are then mapped to *model* states, so-called due to their role in building models of the behavior recorded by the PES.

Using the model states, CHITRA93 allows the user to display the data using several predefined views, including two-dimensional plots, fast Fourier transform plots and a simple textual dump. CHITRA93 also defines a number of useful *transforms* which map a PES to another; these transforms include clipping part of the PES; filtering away all tuples that fall below a certain frequency of occurrence or account for less than a certain amount of time; aggregating all occurrences of a particular pattern of tuples into a single superstate; and projecting one or more elements of the state tuples.

In addition to data display and transformation, CHITRA93 provides several means for *modeling* program behavior as represented by the PES. The first is a semi-Markov model, with output given in terms of a matrix of model-state-to-model-state transition probabilities and a table of model state occupancy time statistics. The second is a model based on decision trees, which extends the semi-Markovian model by considering model state histories that extend past the immediately preceding model state.

As described earlier, our experience with augmenting and maintaining CHITRA93 motivated us greatly to design a software architecture for trace analysis that was optimized for

user augmentation and took into account our concerns regarding tool portability, reliability and maintainability. CHITRA93 had significant deficiencies in all four areas, and as work on CHITRA95 proceeded, so did the evolution and refinement of the aforementioned software architecture and the design principles that underlie it.

1.6 Summary

The problem solved by this thesis is the design and implementation of an easily extensible architecture for the visualization and analysis of trace data. The major innovations brought about in the solution include the blending of the various philosophies of program design to promote extensibility, reusability, portability, reliability and maintainability; the separation of the user interface from code that performs data analysis; and the use of configuration files to specify in an interface-neutral fashion how the system interacts with the user.

1.7 Thesis Overview

Chapter 2 conducts the reader through a brief tour of CHITRA95. Chapter 3 presents the design principles that underlie the software architecture that CHITRA95 implements. Chapter 4 presents the logical design of CHITRA95. Chapter 5 shows how the logical design given by the preceding chapter is actually implemented. Chapter 6 describes the lessons learned from years of continuous use and refinement. Chapter 7 gives our conclusions and suggests directions for further research.

Chapter 2

A BRIEF TOUR OF Chitra95

In this chapter, the family of commands that currently comprise CHITRA95 is illustrated and naming conventions are explained. Several key elements of CHITRA95 are then demonstrated by walking the reader through a mock analysis session. This demonstration is by no means exhaustive; refer to the CHITRA95 User Manual [2] for in-depth information.

2.1 The Chitra95 Command Set

CHITRA95 is composed of a number of *module groups*, each of which contains one or more executable modules which communicate either through CHITRA95's canonical format (i.e., `CFStream`) or through an ensemble in persistent (i.e., disk-based) form. There are eight module groups:

- The *assistants* group, consisting of translators that convert various trace file formats to canonical format, an inserter that converts data in canonical format into an ensemble in persistent form, and assorted helper modules;
- The *extractors* group, consisting of modules that extract data from an ensemble in persistent form and outputs the data in canonical format;
- The *generators* group, consisting of modules that perform various computations on their input and generate their output in canonical format;
- The *modelers* group, consisting of modules that create assorted analytical models of their input;

- The *synthesizers* group, consisting of modules that create synthetic traces based upon some input model;
- The *testers* group, consisting of modules that apply statistical tests to their input;
- The *transformers* group, consisting of modules that apply some form of transformation upon a designated ensemble, usually in persistent form; and
- The *viewers* group, consisting of modules that display their input in some nicely-formatted fashion.

Figure 2.1 lists the various modules that belong to each module group. Each module name consists of terms connected by dots. The first term is always a verb. The last term is always a noun. The other terms, if any, always contain the object of a prepositional phrase (denoted *oopp*). To be specific, the possible forms are:

- *verb.noun* (e.g., `g.ots` — "generate occupancy time statistics")
- *verb.oopp.noun* (e.g., `a.o.csl` — "assist by opening .csl files")
- *verb.oopp.oopp.noun* (e.g, `x.a.fil.ens` — "transform by aggregation by filtering")

Table 2.1 gives a list of the more commonly used modules and what their names mean.

2.2 How Chitra95 Modules Communicate

All CHITRA95 modules communicate either through an ensemble in persistent form or, more commonly, through a canonical format called `CFStream`. `CFStream` implements the canonical format as a stream of text data; this permits modules to read from standard input and write to standard output. More importantly, the use of standard input and standard output allows modules to be used in a pipeline. This practice has the added benefit of encouraging the factoring out of common functionality into separate, reusable modules; furthermore, it allows users to take advantage of popular text-processing programs such as

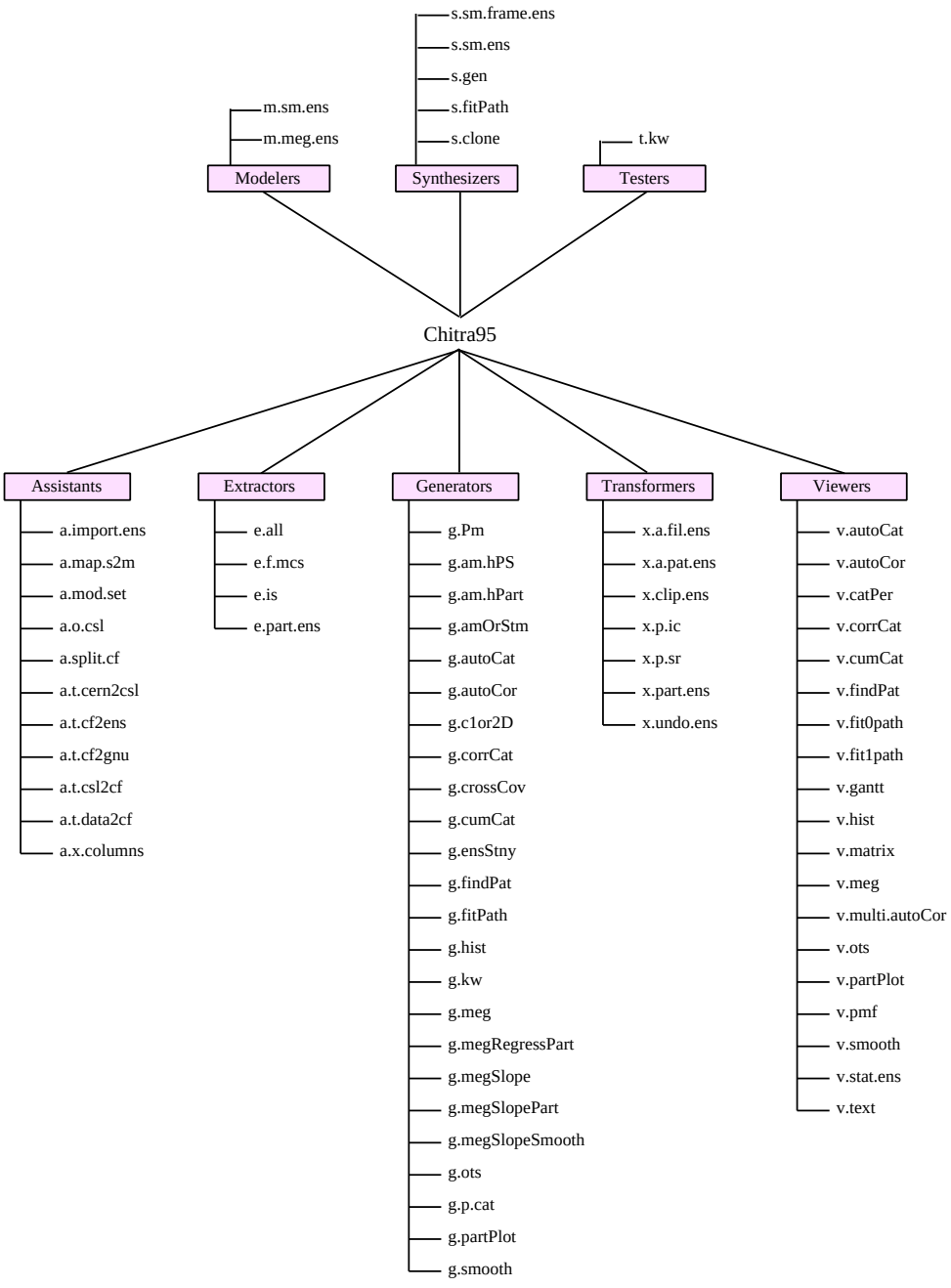

Figure 2.1: The CHITRA95 Command Set

Module	*Meaning of module name*	*Description*
`a.import.ens`	Assist by importing into an ensemble	Imports trace data in various formats into an ensemble in persistent form
`a.t.cf2ens`	Assist by translating canonical format into an ensemble	Converts trace data in canonical format into an ensemble in persistent form
`a.t.data2cf`	Assist by translating data into canonical format	Converts a number of text data formats into canonical format
`e.all`	Extract all	Extracts all or selected information contained in an ensemble in persistent form to standard output in canonical format
`g.amOrStm`	Generate an adjacency matrix or a state transition matrix	Creates one of the two from its input; the result is sent in canonical format to standard output
`g.ots`	Generate occupancy time statistics	Computes various statistics for the occupancy times found in its input
`m.sm.ens`	Model by semi-Markov an ensemble	Creates a semi-Markov model of an ensemble
`s.sm.ens`	Synthesize (a trace) from a semi-Markov model of an ensemble	Creates a synthetic trace derived from a semi-Markov model of a specified ensemble
`t.kw`	Test by Kruskal-Wallis	Tests if same distribution underlies elements of two or more trace files
`x.a.fil.ens`	Transform an ensemble by filter aggregation	Searches an ensemble for sequences of trace records that satisfy some filtering criteria; each sequence found is collapsed into an aggregate record
`x.undo.ens`	Transform by undoing an ensemble	Rescinds one or more transformations applied to an ensemble
`v.gantt`	View a Gantt graph	Displays a Gantt graph of a member of an ensemble
`v.text`	View text dump	Displays a nicely-formatted text dump of the contents of an ensemble

Table 2.1: Some Commonly-Used CHITRA95 Modules

```
0,0
1,1
2,2
3,3
4,5
5,0
6,1
7,2
8,3
9,1
10,0
11,0
```

Figure 2.2: A Sample Trace File

awk, *sed*, *grep* and *perl*, all of which communicate via standard input and standard output. On the other hand, the text-based canonical format does impose upon each module that uses it the cost of translating from text to a binary representation and back, and the verboseness of a textual representation has a measurable impact on efficiency and performance. CHITRA95 modules that require the additional efficiency and performance rely on using the persistent form of an ensemble because this form is directly manipulable; the transformers, in particular, fall into this category.

2.3 A Sample Trace

For the purposes of this demonstration, we make use of a short, artificial trace whose properties may be easily determined by visual inspection. Figure 2.2 gives the trace in the comma-separated value format supported by many commercial spreadsheet packages (e.g., Lotus 1-2-3). This trace contains 12 trace records which are timestamped with absolute times that start at 0 and increase by one time unit with every succeeding trace record. Aside from the timestamp, each trace record has only one other field; the field is numeric and ranges in value from 0 to 5. For example, record 0 consists of timestamp 0 and field value 0; record 7 consists of timestamp 7 and field value 2.

Let this trace be contained in the file `demo.csv`.

2.4 Ensemble Creation

As discussed elsewhere in this thesis, a trace is converted into a member of an ensemble by means of an inserter such as `a.t.cf2ens`. This inserter expects a CHITRA95 canonical format stream (i.e., `CFStream`) as input; the demo trace file must be converted from comma-separated format to canonical format. The conversion is performed by means of the translator `a.t.data2cf`. The command pipeline

```
a.t.data2cf demo.csv | a.t.cf2ens demo.ens
```

translates the demo trace file into canonical format, which is then passed to the inserter which in turn creates the ensemble `demo.ens`, if necessary, and appends the trace data as a new member of the ensemble. Equivalently, the same operation may be performed with the command

```
a.import.ens -o demo.ens demo.csv
```

2.5 Ensemble Output

Once the trace has been converted into an ensemble, it may be output in canonical format by means of an extractor, typically `e.all`. Figure 2.3 shows the output of the command

```
e.all -suppressStateMap -absoluteTime demo.ens
```

The `"-suppressStateMap"` option prevents `e.all` from including information about the ensemble state map (otherwise known as the trace record/trace record encoding map) in the output. The `"-absoluteTime"` option directs `e.all` to emit any timestamps in the form of absolute times instead of the default of occupancy times; if this option had not been specified, `e.all` would have automatically compensated for the inability to compute the occupancy time of the last trace record (recall that the original trace contained absolute times) by dropping the final trace record, as shown in Figure 2.4.

```
##This is a Chitra CF (Canonical Format) file.
#cfSpecNumber=10
#createdOn=Mon Apr  8 12:19:49 1996
#receivedBy=e.all -suppressStateMap -absoluteTime demo.ens
#isCategorical=yes
#maxOfMaxMState=4
#maxOfNumUniqueMStates=5
#memberList=0
#numObjs=1
#srcEnsembleFName=demo.ens
#version=0
#endTime.0=11
#intrinsicTypeOf1.0=uinteger
#intrinsicTypeOf2.0=uinteger
#maxMState.0=4
#numTuples.0=12
#numUniqueMStates.0=5
#objType.0=sequence
#srcMemberFName.0=demo.ens/000000.mbr
#srcMemberID.0=0
#srcTraceFName.0=demo.csv
#startTime.0=0
#timeIncrement.0=0
#tupleLength.0=2
#typeOf1.0=modelState
#typeOf2.0=absoluteTimestamp
0 0
1 1
2 2
3 3
4 4
0 5
1 6
2 7
3 8
1 9
0 10
0 11
#$
```

Figure 2.3: Output of `e.all` with absolute timestamps

```
##This is a Chitra CF (Canonical Format) file.
#cfSpecNumber=10
#createdOn=Mon Apr  8 12:23:20 1996
#receivedBy=e.all -suppressStateMap demo.ens
#isCategorical=yes
#maxOfMaxMState=4
#maxOfNumUniqueMStates=5
#memberList=0
#numObjs=1
#srcEnsembleFName=demo.ens
#version=0
#endTime.0=11
#intrinsicTypeOf1.0=uinteger
#intrinsicTypeOf2.0=uinteger
#maxMState.0=4
#numTuples.0=11
#numUniqueMStates.0=5
#objType.0=sequence
#srcMemberFName.0=demo.ens/000000.mbr
#srcMemberID.0=0
#srcTraceFName.0=demo.csv
#startTime.0=0
#timeIncrement.0=0
#tupleLength.0=2
#typeOf1.0=modelState
#typeOf2.0=occupancyTime
0 1
1 1
2 1
3 1
4 1
0 1
1 1
2 1
3 1
1 1
0 1
#$
```

Figure 2.4: Output of `e.all` with occupancy times

```
================================================================================
Content of member 0 (trace file demo.csv):

      Event    Absolute       Model
     Number        Time       State
================================================================================
          0           0           0
          1           1           1
          2           2           2
          3           3           3
          4           4           4
          5           5           0
          6           6           1
          7           7           2
          8           8           3
          9           9           1
         10          10           0
         11          11           0
```

Figure 2.5: Output of `v.text`

Normally, however, one of the supplied viewers is used to output an ensemble's contents instead of an extractor. For example, Figure 2.5 shows the output from the command

```
v.text demo.ens
```

A Gantt graph of the same ensemble may be displayed to the screen with the command

```
v.gantt -time demo.ens
```

Alternatively, the graph may be created and sent to an Encapsulated PostScript (EPS) file named *demo.eps* with the command

```
v.gantt -time -eps demo.eps monochrome 5,3.5 demo.ens
```

Figure 2.6 shows the graph created by this command.

2.6 Model Generation

One of the modelers provided by CHITRA95 creates and displays a semi-Markov model of an ensemble. Figure 2.7 shows the output from the command

```
m.sm.ens demo.ens
```

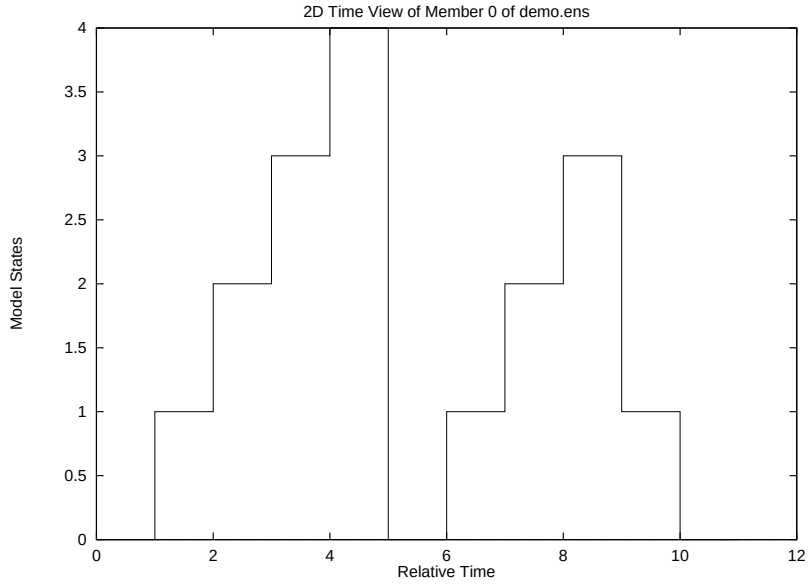

Figure 2.6: Graph created by `v.gantt`

The modeler `m.sm.ens` is actually a script that makes use of four other CHITRA95 modules: `e.all` to read the data from an ensemble; `g.amOrStm` to generate a state transition matrix in canonical format; `v.matrix` to display the matrix in the manner shown above; and `v.stat.ens` to compute and display the table of occupancy time statistics that ends the modeler's display.

2.7 Trace Synthesis and Model Validation

`m.sm.ens` begins by piping the contents of an ensemble extracted with `e.all` to the generator `g.amOrStm`. Instead of displaying the state transition matrix generated by `g.amOrStm`, the latter's canonical format output may instead be piped to the synthesizer `s.sm.ens`, which then creates a synthetic trace; the synthetic trace can then be converted into a new ensemble with `a.t.cf2ens`. The following command pipeline accomplishes all this:

```
e.all -suppressStateMap -suppressTime demo.ens
    | g.amOrStm
    | s.sm.ens -s 0 -r 1 -l 1000 -t 2
    | a.t.cf2ens demo2.ens
```

```
===== Semi-Markov Model of ensemble demo.ens =====
===== Transition probability matrix of discrete time, embedded Markov chain:

Divide the numbers in the table below by 100 to obtain probabilities.
Asterisks in the table represent probabilities smaller than 0.

=========================
        0   1   2   3   4
=========================
   0:     100
   1:..33......67........
   2:             100
   3:......50..........50
   4: 100


===== Model state occupancy time statistics
===== of ensemble demo.ens

================================================================================
Model Num. of      %    Min    Max      Sample      Sample Coefficient      %
State Samples Number    Dur    Dur        Mean      StdDev   Variation   Mean
================================================================================
    0       3 27.273      1      1      1.0000      0.0000      0.0000 27.273
    1       3 27.273      1      1      1.0000      0.0000      0.0000 27.273
    2       2 18.182      1      1      1.0000      0.0000      0.0000 18.182
    3       2 18.182      1      1      1.0000      0.0000      0.0000 18.182
    4       1  9.091      1      1      1.0000         N/A         N/A  9.091

N/A = not applicable
```

Figure 2.7: Output of `m.sm.ens` for ensemble `demo.ens`

```
Divide the numbers in the table below by 100 to obtain probabilities.
Asterisks in the table represent probabilities smaller than 0.

========================
        0   1   2   3   4
========================
   0:     100
   1:..32......68........
   2:             100
   3:......45.........55
   4: 100
```

Figure 2.8: Output of `v.matrix` for ensemble `demo2.ens`

Passing the `"-suppressTime"` option to `e.all` suppresses timestamp information that `g.amOrStm` does not recognize. The options to `s.sm.ens` tell it to begin the synthetic trace at model state 0, use a random seed of 1, create a trace with 1000 records, with each record containing both a model state and a timestamp (i.e., tuple length of 2). The trace is then converted into a member of the ensemble `demo2.ens`.

Model validation can then be performed by comparing the transition probability matrix that characterizes `demo2.ens` to that of the original ensemble `demo.ens`. Figure 2.8 shows the output from the following command pipeline:

```
e.all -suppressStateMap -suppressTime demo2.ens
    | g.amOrStm
    | v.matrix
```

It can be seen that the matrix for `demo2.ens` contains transitions that match each of those in the matrix for `demo.ens` and that corresponding values are within a few percentage points of each other; from this, it can be concluded that the semi-Markov model created by `g.amOrStm` from an ensemble is a fair representation of the behavior captured by the ensemble.

2.8 Version Control

CHITRA95 supports the maintainance of multiple versions of an ensemble. When initially constructed, an ensemble is at version 0. Ensemble version k results from the appli-

```
Member Total number  Number of   % States matched
    Id    of States    Matches         by pattern
================================================
     0           12          2              41.67

Made 2 match(es) totalling 41.67% of the original ensemble.
```

Figure 2.9: Output of `x.a.fil.ens`

cation of any number of modifications to ensemble version $k-1$. Modifications made to an ensemble may be undone by applying to it the *undo* operation. In general, up to k levels of undo may be applied to an ensemble at version k.

For example, the following command applies a *filter aggregation transform* to the ensemble `demo.ens`:

```
x.a.fil.ens -timeDomain demo.ens 25
```

This command aggregates those trace records whose total duration accounts for less than 25shows the output from the `x.a.fil.ens` module; Figure 2.10 shows the contents of the ensemble after the application of the transform. Note that the ensemble version has been incremented; it now stands at one. Each subsequent transformation will increment the ensemble version.

If the modifications made by the transform are unsatisfactory or otherwise undesirable, they made be undone with the command

```
x.undo.ens demo.ens
```

This command rescinds all modifications made to arrive at the current ensemble version. The ensemble then reverts to its state prior to the application of the transform.

2.9 A Note About Chitra95's Canonical Format

A file in CHITRA95 canonical format is an ASCII text file containing *records* of three distinct types: *attribute records*, *control records* and *data records*. A record consists of a single line of data; the newline character terminates the record. Attribute and control

```
##This is a Chitra CF (Canonical Format) file.
#cfSpecNumber=10
#createdOn=Mon Apr 22 15:14:56 1996
#receivedBy=e.all -absoluteTime -suppressStateMap demo.ens
#isCategorical=yes
#maxOfMaxMState=6
#maxOfNumUniqueMStates=4
#memberList=0
#minOfMinMState=0
#numObjs=1
#srcEnsembleFName=demo.ens
#version=1
#endTime.0=11
#intrinsicTypeOf1.0=uinteger
#intrinsicTypeOf2.0=uinteger
#maxMState.0=6
#numTuples.0=5
#numUniqueMStates.0=4
#objType.0=sequence
#srcMemberFName.0=demo.ens/000000.mbr
#srcMemberID.0=0
#srcTraceFName.0=demo.csv
#startTime.0=0
#timeIncrement.0=0
#tupleLength.0=2
#typeOf1.0=modelState
#typeOf2.0=absoluteTimestamp
0 0
5 2
0 5
6 7
1 9
#$
```

Figure 2.10: The ensemble after application of the filter aggregation transform

records have a `'#'` character in the first column; records without this prefix are data records. Comment records begin with `"##"`. Figure 2.3 shows a typical canonical format file.

Data records are organized into *objects*. A canonical format file contains data for N distinct objects separated by the end-of-object marker `"#$"`. Each data record contains M components numbered from 1 to M where M may vary from object to object; however, all data records within the same object have the same number of components. A canonical format file starts with attribute records; the data records for the first object follow immediately afterwards. An end-of-object marker separates the last data record of object 0 and the first data record of object 1.

An attribute record consists of an *attribute name*, an optional *object number* and an *attribute value*. If no object number is specified, the attribute is *global*. In the example, the global attribute `version` has the value 0, while the `objType` attribute associated with object 0 has the value `sequence`. Table 2.2 describes each of the global attributes appearing in the canonical format file shown in Figure 2.3; Table 2.3 describes each of the object-specific attributes appearing in the same figure.

Attribute name	*Description*
`cfSpecNumber`	The revision number of the canonical format specification used to generate the file
`createdOn`	Time and date the file was created
`isCategorical`	Indicates whether or not the data contained in the file is categorical
`maxOfMaxMState`	Maximum-valued model state appearing in any object in the file denoting an ensemble member
`maxOfNumUniqueMStates`	Maximum number of unique model states among all objects in the file that denote an ensemble member
`memberList`	List of member IDs corresponding to the ensemble member objects in the file
`numObjs`	Number of objects appearing in the file
`receivedBy`	The command line that invoked the module that created the file; this record is called a *receipt* record
`srcEnsembleFName`	The name of the ensemble from which the data appearing in the file is derived
`version`	The version of the ensemble from which the data appearing in the file is derived

Table 2.2: Common Global Attributes

Attribute name	*Description*
`endTime`	The ending time of the ensemble member from which the object is derived
`intrinsicTypeOf`k	The basic data type of the kth component of a data record belonging to the object
`maxMState`	The maximum-valued model state in the ensemble member from which the object is derived
`numTuples`	The number of data records in the object
`numUniqueMStates`	The number of unique model states in the ensemble member from which the object is derived
`objType`	The type of the object
`srcMemberFName`	The name of the ensemble member from which the object is derived
`srcMemberID`	The numeric ID of the ensemble member from which the object is derived
`srcTraceFName`	The name of the trace file from which the ensemble member denoted by the object was originally derived
`startTime`	The starting time of the ensemble member from which the object is derived
`timeIncrement`	If greater than zero, the amount of time between any two consecutive data records
`tupleLength`	The number of components in each data record in the object
`typeOf`k	The logical data type of the kth component of a data record belonging to the object

Table 2.3: Common Object-Specific Attributes

Chapter 3
DESIGN PRINCIPLES

The design of CHITRA95 follows a number of design principles which were derived from our experience with earlier CHITRA versions as well as other software systems for trace analysis and data visualization. As such, the principles apply not only to the design of CHITRA95 but also to the design of any future trace analysis tool. These principles may be summarized as follows:

- Use the toolkit approach to software construction
  - Propagate object orientation throughout the design.
  - Factor out common operations.
  - Prepare to be assimilated.
- Separate data visualization from data generation.
  - Expect new viewers to be added.
- Automate user interface construction.
- Save time and effort with non-interactive batch testing.

3.1 Use the toolkit approach to software construction.

Two common approaches to writing software are the *monolithic* approach and the *modular* approach. However, most software designers choose between these approaches in the context of writing a single, integrated program, as opposed to CHITRA95's use of multiple, small modules. CHITRA95 takes the module approach further by organizing its modules as

a toolkit of reusable parts from which an integrated application may be built. Building a toolkit of parts has the added benefit of facilitating the creation of a specialized turnkey application by allowing the application builder to take only what is needed.

3.1.1 Propagate object orientation throughout the design.

The object-oriented model focuses on data and the operations that may be applied to that data. Often, system designers stop at the program level when applying object orientation. A trace visualization and analysis system is fundamentally a data repository plus a set of viewing and analysis operations, making the object-oriented model well suited to the design of such a system. Isolate into separate modules the maintainance of the data repository from the operations (i.e., "methods") that apply to the data; doing so makes it easier to add new operations in the future.

3.1.2 Factor out common operations.

Many visualization and analysis tasks share common operations. For example, a family of two-dimensional graph viewers may share the same sequence of data preprocessing operations, such as argument parsing. By factoring out these common operations into separate modules, the task of writing an additional graph viewer of the same family is simplified. Moreover, all viewers in the family benefit from bug fixes in and enhancements to the shared code.

3.1.3 Prepare to be assimilated.

Many software tool designers commit the mistake of assuming that users of the tool will always want to augment the tool, as opposed to wanting to incorporate the tool's desirable features into *other* similar tools. Allow for modules to be reused, possibly by third-party trace visualization and/or analysis systems, by making as few assumptions as possible with regard to the modules' input, output and interface requirements. Just as importantly, be consistent with any assumptions that you do make; this simplifies the task of assimilation.

Judicious layering such as that used in the software architecture of CHITRA95, coupled with appropriate inter-module translation, can mask implementation details when necessary.

3.2 Separate data visualization from data generation.

Data visualization and data generation should be separated into their own modules. This permits new visualizers to be dropped into the system without making undue impact on existing data generators. At the same time, it permits new generators to be developed rapidly, particularly when the data they generate can be viewed with an existing module.

3.2.1 Expect new viewers to be added.

Insight into the behavior of a system from which a trace or set of traces is derived is sometimes the result of applying novel visualization methods to the trace(s). For example, cyclic behavior may become more evident when viewed in polar or spherical coordinates. In general, the designer of a trace analysis tool cannot assume that the set of viewers she provides will be adequate for the needs of all of the tool's users; instead, she must assume and prepare for the addition by users of new viewers. Few, if any, assumptions should be made regarding the nature and use of these new viewers. For example, the tool designer should not assume that the viewer will be part of a standard, windowed graphical user interface (e.g., assume the availability of menus and dialog boxes).

3.3 Automate user interface construction.

One of the more daunting tasks facing the user wishing to write a new module for any software system is having to integrate the module into the system's existing user interface. Often, the user is either unable or unwilling to expend the necessary time and effort; as a result, either the module is never written at all or the coherence and consistency of the system suffers from imperfect integration. Automate the generation of the associated user interface by the use of external configuration files that specify content as opposed to

behavior. This permits the interface to be tailored for different user groups, as well as facilitating the rapid construction and deployment of a turnkey, integrated trace analysis and/or visualization tool.

3.4 Save time and effort with non-interactive batch testing.

Any tool for trace visualization and analysis is as useful only as far as its results can be relied upon. The process of building such a tool is always punctuated by the need for tool validation and verification. CHITRA95's architecture allows modules to be validated individually; confidence in CHITRA95 can therefore be built one module at a time. The use of standard input and standard output permits modules to be verified simply by writing scripts that test their output against known samples. Non-interactive batch testing provides results that are easy to obtain and, best of all, repeatable.

Chapter 4

THE DESIGN OF Chitra95

This chapter presents the logical structure of CHITRA95, first on a gross level, followed by discussion of the individual modules that comprise the design.

4.1 Logical Structure

In accordance with the design principles listed in the preceding section, the architecture of CHITRA95 is composed of numerous small modules which are organized into several well-defined groups. An illustration of CHITRA95's logical structure appears in Figure 4.1.

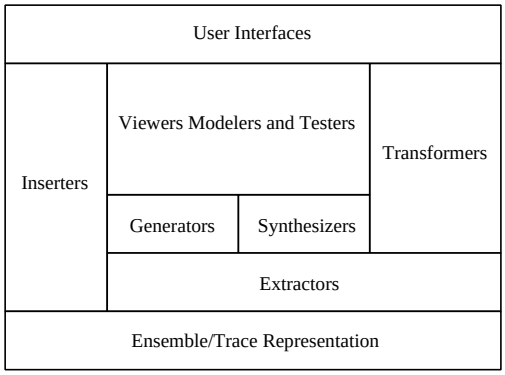

Figure 4.1: Logical Structure of CHITRA95

At the bottom lies the *ensemble representation*, which in turn contains one or more instances of the *trace representation*. The ensemble and trace representations are *persistent*; in other words, their lifetime may span the boundaries between one or more analysis sessions. Each trace in an ensemble is also referred to as an ensemble *member*.

Atop the bottom layer is another one composed of *data inserters and extractors*. These

modules are the only ones which interact directly with the ensemble representation. Beyond the inserter/extractor layer is the layer of generators, transformers and synthesizers. In the context of CHITRA95, a generator is any module which takes ensemble data and performs some kind of analytical operation upon it such as generating a model. A transformer is any module that takes ensemble data and modifies it in some fashion, typically with the goal of either reducing or partitioning the data. A synthesizer is a module that creates a synthetic trace. No generator, transformer or synthesizer deals directly with the ensemble representation; instead all make use of the inserter/extractor layer to access the ensemble data. It should be clear that the inserter/extractor layer factors out all input/output code with respect to the ensemble representation from all generators and transformers, thus minimizing code duplication and simplifying the task of debugging input/output-related problems.

Atop the generator/transformer/synthesizer layer is the layer containing *viewers, modelers and testers*, also collectively known as *presenters*. A presenter is a module that takes the result of an extractor, a generator or a synthesizer and formats the result in a fashion suitable for viewing and/or interaction. A presenter may also create a model or perform a statistical test, the output for which is then delivered to another presenter for final formatting. For example, one of the more often used presenters takes ensemble data and prints it in the form of an easy-to read table, complete with informative headings. In the same way that ensemble input/output is factored out of the generator/transformer/synthesizer layer, generation, transformation and synthesis tasks are factored out of the presenter layer.

Finally, at the top is the layer containing *user interfaces*. A user interface provides an organized framework to underlying layers, allowing the user to perform ensemble input, output, analysis and transformation. It primarily makes use of the presenters and may modify their output to satisfy some aesthetic criteria. CHITRA95 modules are all capable of being invoked at the command line, which is how a user interface invokes them. The modules' *appearance* is automatically determined by the user interface in conjunction with configuration files that specify the *content* of each module's portion of the user interface.

Work is currently underway on a number of interfaces for CHITRA95, including one with a traditional window-based appearance implemented using the new Java programming language and another based upon the forms interface popularized by HTML documents on the World-Wide Web.

The issue of module verification is addressed with the use of test suites that compare module output with known samples. CHITRA95 subjects each major module to a battery of tests, both individually and as part of various series of operations. The layered design also simplifies testing by allowing each individual test to build upon the confidence generated at lower levels of testing.

4.1.1 Terminology and Notational Conventions

The data maintained by the trace and ensemble representations, in addition to the operations that may be performed upon them, are described with the help of the following terminology and notation adapted from [3].

The entity or system from which a trace is obtained is called the *target* and is described by a set of *trace records* denoted by $\mathcal{R}$. A trace record consists of one or more fields that collectively describe the target system over some interval of time. For example, the trace record in a communication network might be an ordered pair *(host-name, length of packet last injected)*. The composition of the trace record is governed by the trace study objective.

Let Z denote the set of non-negative integers. A *trace of length* L is a sequence of trace records R_0, R_1, ..., R_i, ..., R_{L-1}, where $0 \leq i < L$. Optionally all trace records may be associated with a timestamp, in which case we represent $R_i = (\theta_i, r_i)$ where $\theta_i = \theta(R_i)$ is the timestamp associated with R_i and r_i consists of all the fields of trace record R_i with the exception of the timestamp. Hence, we also write the sequence as (θ_0, r_0), (θ_1, r_1), ..., (θ_i, r_i), ..., (θ_{L-1}, r_{L-1}), where $\langle \forall i : 0 \leq i < L :: \theta_i \in Z \wedge r_i \in \mathcal{R} \rangle$. In general, the trace is associated with a number of attributes, including the type of its timestamps (either *absolute time* or *occupancy time*), if any; the trace's starting time; and the trace's ending time. The timestamp θ_i may denote an absolute point in time, in which case $\langle \forall i : 0 < i < L :: \theta_{i-1} \leq$

$\theta_i\rangle$. Alternately, θ_i may denote an occupancy time, in which case the timestamps increase monotonically $\langle\forall i : 0 < i < L :: \theta_i \geq 0\rangle$. Each pair of successive trace records (R_j, R_{j+1}) (for $0 \leq j < L - 1$) is a *transition*. An *event* causes a transition; the *event number* of R_i is i. An *ensemble* is a set of traces with identically defined trace records, each representing a different observation of the target. An ensemble *member* is one trace in the ensemble; each ensemble member is identified by a unique, small non-negative integer. In an ensemble instance $\mathcal{E}$ containing N members, the kth member is denoted $\mathcal{E}.k$ and the set of trace records that occur in $\mathcal{E}.k$ is denoted $\mathcal{R}(\mathcal{E}.k)$.

Each ensemble instance $\mathcal{E}$ contains one or more *versions*, denoted $\mathcal{E}_0$, $\mathcal{E}_1$, ..., $\mathcal{E}_\upsilon$, ... The version designated $\mathcal{E}_0$ is that which exists immediately after the ensemble is first created; successive versions may be started at any time. All modifications made to an ensemble since the current or a preceding version was started may be rescinded with an *undo* operation. In general, up to υ levels of undo may be applied to ensemble $\mathcal{E}_\upsilon$.

Since any operation applied to the ensemble may potentially modify its record space, the term $\mathcal{R}(\mathcal{E}_\upsilon)$ denotes the set of all trace records that appear in all traces in ensemble version $\mathcal{E}_\upsilon$. CHITRA95 maps the elements of $\mathcal{R}(\mathcal{E}_\upsilon)$ to non-negative integers called *trace record encodings* (*encoding* for short). The function $X_{\mathcal{E}_j}(y)$ maps y to Z, where the domain of y includes all $r \in \mathcal{R}$, Z, any arbitrary set $\{m|m \in Z\}$ and any arbitrary sequence $(m_0, m_1, \ldots, m_i, \ldots)$ where $\langle\forall i : m_i \in Z\rangle$. For example, $X_{\mathcal{E}_j}(r_i)$ denotes the encoding that trace record r_i maps to in ensemble $\mathcal{E}_j$; similarly, $X_{\mathcal{E}_j}(m)$ denotes the encoding that trace record encoding m maps to in $\mathcal{E}_j$. The initial set of mappings $\mathcal{X}_{\mathcal{E}_0}$ is generated by CHITRA95 when the ensemble is created.

4.2 The Trace Representation

Our experience with trace analysis shows that a trace representation needs to support a set of simple primitive operations. These operations are given in Table 4.1. The first four operations (i.e., insertion, deletion, appending, and sequence collapsing) are typically

Operation	*Description*
insert	Insert a trace record at a specified position
delete	Delete the trace record at a specified position
append	Append a new trace record
collapse-subsequence	Replace a specified subsequence of trace records with a new trace record
get-length	Returns the number of trace records currently contained in the representation
get-record	Retrieves the trace record at a specified position
get-attribute	Retrieves the value of some attribute associated with the trace, including the timestamp type, starting time and ending time of the trace

Table 4.1: Primitive Trace Representation Operations

used while building up the trace representation from one or more trace files. The other operations, which do not modify the trace representation, are used in conjunction with any method that explores the data represented. Any other operations, particularly those that compute assorted statistics or sets of values, can be defined using these primitive operations.

In CHITRA95, the trace file (θ_0, r_0), (θ_1, r_1), ...,(θ_{L-1}, r_{L-1}), is internally represented as a timestamp sequence θ_0, θ_1, ..., θ_{L-1} and a trace record encoding sequence M_0, M_1, ..., M_{L-1} where $M_i = X_{\mathcal{E}_v}(r_i)$. CHITRA95 does not directly store a sequence of trace records, primarily for efficiency of representation but also for the benefit of model generation and other cases where the user desires a one-dimensional view of multi-field trace records. The data is maintained in an ensemble member instance which provides access via a number of simple operations; these operations vary from those listed in Table 4.1 to take the internal representation into account. Additionally, several operations are provided that yield important aggregate and statistical information about the data contained by the member; these operations are included in the ensemble member's interface as an optimization.

Each ensemble member maintains a record of its starting time, its ending time and its timestamp type; in particular, the starting time and ending time of the member are updated

to reflect modifications to the head and tail, respectively, of the ensemble member. The *current length* L of the ensemble member is defined as the number of trace records inserted and appended minus the number of trace records deleted (the *collapse* operation consists of one or more *delete* operations followed by an *insert* operation). In the following discussion, all references to L refer to the length of the ensemble member that is current immediately prior to the application of some operation.

The primitive operations given in Table 4.1 are decomposed into actual ensemble member operations; the latter are summarized in Table 4.2; they are formally defined as follows:

- The *insert* operation inserts the ordered pair (θ', M') into the member at position k where $0 \leq k < L$. That is, if the member prior to the operation is given by (θ_0, M_0), (θ_1, M_1), ...,(θ_{L-1}, M_{L-1}), the modified member is given by (θ_0, M_0), (θ_1, M_1), ..., (θ_{k-1}, M_{k-1}), (θ'_k, M'_k), (θ_{k+1}, M_{k+1}), ..., (θ_L, M_L).

- The *delete* operation deletes the ordered pair (θ', M') from the member at position k where $0 \leq k < L$. That is, if the member prior to the operation is given by (θ_0, M_0), (θ_1, M_1), ..., (θ_{k-1}, M_{k-1}), (θ'_k, M'_k), (θ_{k+1}, M_{k+1}), ..., (θ_{L-1}, M_{L-1}), the modified member is given by (θ_0, M_0), (θ_1, M_1), ..., (θ_{k-1}, M_{k-1}), (θ_{k+1}, M_{k+1}), ..., (θ_{L-2}, M_{L-2}).

- The *append* operation appends the ordered pair (θ', M') to the member. That is, if the member prior to the operation is given by (θ_0, M_0), (θ_1, M_1), ...,(θ_{L-1}, M_{L-1}), the modified member is given by (θ_0, M_0), (θ_1, M_1), ..., (θ_{L-1}, M_{L-1}), (θ', M').

- The *collapse-subsequence* operation takes a subsequence of the member with length n starting at position k and replaces it with a single ordered pair (θ', M') where either $\theta' = \theta_k$ if the member's timestamps are absolute times or $\theta' = \sum_{i=k}^{k+n-1} \theta_i$ if the member's timestamps are occupancy times; the operation automatically computes the proper value of θ' with the aid of the *get-timestamp-type* operation. M' is an *aggregate trace record encoding* that replaces the trace record encoding sequence M_k,

Operation	*Description*
insert	Inserts the trace record (θ, M) into position k
delete	Deletes the trace record (θ_k, M_k) from the member
append	Appends the trace record (θ, M) to the member
collapse-subsequence	Replaces n trace records starting at position k with the new trace record (θ, M)
get-length	Returns the current length of the member
get-record:	
get-encoding	Returns the value of M_k
get-absolute-timestamp	Returns the absolute timestamp associated with the trace record in position k
get-occupancy-time	Returns the occupancy time associated with the trace record in position k
get-attribute:	
get-timestamp-type	Returns the type of the timestamps contained in each trace record in the member
get-starting-time	Returns the starting time of the member
get-ending-time	Returns the ending time of the member
get-set-of-unique-trace-records	Returns the set of unique trace records $\{r\}$ contained in the member
get-set-of-unique-encodings	Returns the set of unique trace record encodings $\{m\}$ contained in the member
get-maximum-encoding	Returns the maximum-valued trace record encoding contained by the member
get-number-of-unique-encodings	Counts the number of unique trace record encodings contained in the member
count-encoding-occurrences	Counts the number of times a specific trace record encoding appears in the member
check-if-encoding-is-contained	Determines if the member contains a specific trace record encoding
get-encoding-duration	Determines the total amount of time accounted for in the member by a specific trace record encoding

Table 4.2: Ensemble Member Operations

$M_{k+1}, \ldots, M_{k+n-2}, M_{k+n-1}$ such that $M' \notin \{M_i | 0 \leq i < L\}$. That is, if the member prior to the operation is given by (θ_0, M_0), (θ_1, M_1), $\ldots$, (θ_{k-1}, M_{k-1}), (θ_k, M_k), $\ldots$, (θ_{k+n}, M_{k+n}), $\ldots$, (θ_{L-1}, M_{L-1}), the modified member is given by the sequence (θ_0, M_0), (θ_1, M_1), $\ldots$, (θ_{k-1}, M_{k-1}), (θ'_k, M'_k), (θ_{k+n}, M_{k+n}), $\ldots$, $(\theta_{L-n-1}, M_{L-n-1})$.

- The *get-timestamp-type* operation returns the type of the timestamps θ associated with each ordered pair contained by the member.

- The *get-starting-time* operation returns the starting time T_S of the member. This is simply θ_0 if each θ is an absolute timestamp, or 0 if each θ_i is an occupancy time.

- The *get-ending-time* operation returns the ending time T_E of the member. This is either infinity if each θ_i is an absolute timestamp or the member's starting time plus $\sum_{i=0}^{L-1} \theta_i$ if each θ_i is an occupancy time.

- The *get-encoding* operation returns the trace record encoding M_k associated with the ordered pair in the kth position of the member.

- The *get-absolute-timestamp* operation returns the absolute timestamp associated with the ordered pair in the kth position of the member. If the member's timestamp type is *absolute time*, this operation simply returns θ_k. Otherwise, it returns $T_S + \sum_{i=0}^{k-1} \theta_i$.

- The *get-occupancy-time* operation returns the occupancy time associated with the ordered pair in the kth position of the member. If the member's timestamp type is *occupancy time*, this operation simply returns θ_k. Otherwise, it returns either $\theta_{k+1} - \theta_k$ if $k < L - 1$ or 0 if $k = L - 1$.

- The *get-length* operation returns the current length L of the member.

- The *get-set-of-unique-trace-records* operation returns the set $\{r | r \in \mathcal{R}(\mathcal{E}_v) \wedge r = r_i, 0 \leq i < L\}$ associated with the member.

- The *get-set-of-unique-encodings* operation returns the set $\{m | m \in \mathcal{X}_{\mathcal{E}_v} \wedge m = M_i, 0 \leq i < L\}$ associated with the member.

- The *get-maximum-encoding* operation returns the maximum-valued trace record encoding in the set of unique encodings contained by the member.
- The *get-number-of-unique-encodings* operation returns the cardinality of the set of unique encodings contained by the member.
- The *count-encoding-occurrences* operation takes as an argument a trace record encoding m and returns the number of times that m appears in the member.
- The *check-if-encoding-is-contained* operation takes as an argument a trace record encoding m and determines whether m occurs at least once in the member.
- The *get-encoding-duration* operation takes as an argument a trace record encoding m and returns the total amount of time accounted for by m in the member.

4.3 The Ensemble Representation

The ensemble instance encapsulates one or more member instances and provides access to the each member's information via the operations listed in §4.2. Additionally, the ensemble supports additional operations of its own that always apply to the current ensemble version $\mathcal{E}_v$. These operations are divided into the following categories: ensemble construction; ensemble-level queries and statistics computation; translation between trace records and trace record encodings; and ensemble version control. Table 4.3 summarizes the operations defined for each ensemble.

4.3.1 Ensemble Construction

Ensembles are constructed one member at a time. The *add-member* operation creates a new, empty ensemble member. If the ensemble contains N members $\mathcal{E}.0$, $\mathcal{E}.1$, ..., $\mathcal{E}.N-1$ prior to the operation, the new member is denoted $\mathcal{E}.N$. Once a member has been added to the ensemble, the operations described in §4.2 may be applied to the member.

Operation	*Description*
add-member	Adds a new, empty member to the ensemble
get-number-of-members	Returns the number of member currently contained by the ensemble
get-length	Returns the current total length of the ensemble
get-set-of-unique-trace-records	Returns the set of unique trace records $\{r\}$ contained by the ensemble
get-set-of-unique-encodings	Returns the set of unique trace record encodings $\{m\}$ contained by the ensemble
get-maximum-encoding	Returns the maximum-valued trace record encoding contained by the ensemble
get-number-of-unique-encodings	Counts the number of unique trace record encodings contained in the ensemble
count-encoding-occurrences	Counts the number of times a specific trace record encoding appears in the ensemble
check-if-encoding-is-contained	Determines if the ensemble contains a specific trace record encoding
get-encoding-duration	Determines the total amount of time accounted for in the ensemble by a specific trace record encoding
map-trace-record	Maps a trace record r to a trace record encoding M
map-single-encoding	Maps a trace record encoding M to a new encoding M'
map-encoding-sequence	Maps a sequence $(M_0, M_1, \ldots)$ of trace record encodings to a new encoding M'
map-encoding-set	Maps a set $\{M_0, M_1, \ldots\}$ of trace record encodings to a new encoding M'
unmap-encoding	Unmaps a trace record encoding M to the trace record, single encoding, encoding sequence or encoding set from which M was originally mapped
get-version	Returns the current version number of the ensemble
start-new-version	Prepares a new version of the ensemble and increments the ensemble's version number
undo	Rescinds all modifications made to one or more versions of the ensemble, starting with the current one

Table 4.3: Ensemble Operations

4.3.2 Ensemble-level Queries and Statistics Computation

The ensemble defines several query and statistics computation operations that parallel a number of similarly-named operations at the ensemble member level; the ensemble-level operations invoke the corresponding member operations for each member belonging to the ensemble and combine the results appropriately. These ensemble operations are as follows:

- The *get-number-of-members* operation returns the current number of members contained by the ensemble.
- The *get-length* operation returns $L_{\mathcal{E}}$ where $L_{\mathcal{E}} = \sum_{i=0}^{N-1} L_{\mathcal{E}.i}$ and N is the number of members currently contained by the ensemble. That is, the length of the ensemble is the sum of the current lengths of its members.
- The *get-set-of-unique-trace-records* operation returns the set $\mathcal{R}(\mathcal{E}_v)$.
- The *get-set-of-unique-encodings* operation returns the set $\{X_{\mathcal{E}_v}(r) | r \in \mathcal{R}(\mathcal{E}_v)\}$.
- The *get-maximum-encoding* operation returns the maximum-valued trace record encoding in the set of unique encodings contained by the ensemble.
- The *get-number-of-unique-encodings* operation returns the cardinality of the set of unique encodings contained by the ensemble.
- The *count-encoding-occurrences* operation takes as an argument a trace record encoding m and returns the number of times that m appears in the ensemble.
- The *check-if-encoding-is-contained* operation takes as an argument a trace record encoding m and determines whether m occurs at least once in the ensemble.
- The *get-encoding-duration* operation takes as an argument a trace record encoding m and returns the total amount of time accounted for by m in the ensemble.

4.3.3 Translation Between Trace Records and Trace Record Encodings

Recall from §4.2 that an ensemble member is internally represented as a sequence of (timestamp, trace record encoding) pairs. The ensemble provides a set of operations for uniquely mapping between trace records and trace record encodings; these operations appear at the ensemble level in order to provide a consistent set of mappings among all members of the ensemble. Additionally, the ensemble also provides a set of operations for uniquely mapping between single trace record encodings and single encodings, sequences of encodings and sets of encodings. These mapping-related operations are as follows:

- The *map-trace-record* operation computes $X_{\mathcal{E}_v}(r)$ for some arbitrary trace record r.
- The *map-single-encoding* operation takes as its argument the trace record encoding m and computes $X_{\mathcal{E}_v}(m)$. This operation is used to provide alternate mappings to compensate for the distortion introduced when mapping categorical data.
- The *map-encoding-sequence* operation computes $X_{\mathcal{E}_v}(s)$ where s is some arbitrary trace record encoding sequence $(M_0, M_1, \ldots, M_i, \ldots)$.
- The *map-encoding-set* operation computes $X_{\mathcal{E}_v}(s)$ where s is some arbitrary trace record encoding set $\{M_0, M_1, \ldots, M_i, \ldots\}$.
- The *unmap-encoding* operation takes an arbitrary encoding m and decodes it back to the entity it was immediately encoded from (i.e. a trace record, another encoding, an encoding sequence or an encoding set).

4.3.4 Version Control

An important feature of the ensemble is the ability to maintain multiple *versions* and to *undo* modifications made to the ensemble since a specified version was started. Ensemble versions behave much like stacks; the current version of the ensemble is the only version available (i.e., simultaneous access to versions other than the current one is not allowed).

Version control permits the user to experimentally transform an ensemble during the course of analysis without the risk of losing trace information. The version-related operations are as follows:

- The *get-version* operation returns the current version number v of the ensemble $\mathcal{E}_v$.
- The *start-new-version* operation records the current state of the ensemble such that all succeeding modifications to the ensemble may be rescinded if necessary or desired.
- The *undo* operation rescinds all modifications made to the ensemble since the start of a specified number of versions were started.

4.4 The Inserter/Extractor Layer

The information contained by a trace is converted into an ensemble member by means of an *inserter*. In CHITRA95, an inserter is any module that takes one or more sequences of trace records (θ_0, r_0), (θ_1, r_1), ..., (θ_i, r_i), ..., (θ_{L-1}, r_{L-1}), where L may be different for each sequence, and converts each sequence into a separate member of one ensemble. The inserter is responsible for converting each trace record into a trace record encoding to conform with the ensemble member's internal representation. CHITRA95 includes two such inserter programs: `a.import.ens` and `a.o.csl`, the latter name meaning "assist by opening (a) CSL trace(s)".

Alternatively, an inserter may also take one or more sequences of ordered pairs (θ_0, M_0), (θ_1, M_1), ..., (θ_i, M_i), ..., (θ_{L-1}, M_{L-1}), where M_i is a trace record encoding $\forall i : 0 \leq i < L$ and L may be different for each sequence. In this case, the inserter assumes that each trace record has been properly encoded; again, each sequence is converted into a separate ensemble member. CHITRA95 includes one such inserter program named `a.t.cf2ens`.

In addition to taking one or more sequences of ordered pairs, an inserter may also store in the ensemble a set of *attributes* related to the sequence or sequences. In this sense, an attribute is some item of information about a specific sequence or the set of sequences

taken as whole that is not otherwise expressed explicitly by the sequence data; examples of CHITRA95 attributes include the name of the original trace file from which a sequence was derived, the value of the largest trace record encoding in a sequence and the number of unique trace record encodings in the set of sequences. CHITRA95 uses a standard *canonical format* to represent an attributed set of streamed sequences; this format is called `CFStream` and is the *lingua franca* of CHITRA95 modules.

In a similar fashion, the information contained by an ensemble can be converted back into ordered pair sequences by means of an *extractor*. In CHITRA95, an extractor is any module that takes an ensemble and outputs one or more sequences of ordered pairs of either of the two forms taken as input by an inserter. The ensemble information is output in the standard canonical format and consists of one or more sequences, each sequence corresponding to an ensemble member, plus the set of attributes corresponding to the sequences, both individually and as a set. The output typically includes a set of sequences that enumerate all known trace record encodings and the trace records they correspond to, as well as detailed information about the modules that constitute each trace record. The most commonly used CHITRA95 extractor, named `e.all` (i.e., "extract all"), includes this among its various capabilities. All CHITRA95 extractors have names of the form `e.<name>`.

Instead of simply extracting the data from one or more ensemble members, an extractor may extract some subset of the data. For example, one of the extractors provided by CHITRA95 called `e.is` (i.e., "extract initial state") outputs only the first trace record encoding from each member of the ensemble. An extractor may also perform some computation upon the ensemble data and output the result in canonical format for consumption by another CHITRA95 module, typically at a higher layer. An example is the `e.f.mcs` (i.e., "extract by finding the most common state") extractor, which takes a list of trace record encodings and outputs a list of ensemble members, each of which contains one or more instances of the encodings.

The fourth CHITRA95 extractor, named `e.part.ens`, partitions a specified member of an ensemble into one or more pieces of equal length (except possibly for the last piece);

the data for each piece is then recast into a separate ordered pair sequence. The output is therefore a new ensemble with each separate sequence constituting a new ensemble member.

4.5 The Generator/Transformer/Synthesizer Layer

On top of the inserter/extractor layer lies the layer containing generators, transformers and synthesizers.

4.5.1 Generators

A *generator* is a CHITRA95 module that typically takes one or more sequences of ordered pairs as described in §4.4 in canonical format, upon which it performs some kind of analytical operation (e.g., computing statistics, computing the number of bins and bin sizes for a histogram), the result of which is output in canonical format as well. However, generators are not restricted to emitting only ordered pair sequences as input or output; the canonical format allows for other data such as sequences of simple numbers (i.e. 1-tuples) or data describing a matrix. Generators never display their results; instead, the results are always passed to a viewer, a tester or a modeler for presentation.

CHITRA95 generators have names of the form `g.<name>`; they include the following:

- `g.am.hPS` reads the output of `g.amOrStm`; the adjacency matrix given in the latter is searched for a trace record encoding that may be used to cluster the ensemble into smaller homogeneous ensembles; if found, the encoding is then written to output. This generator is used by the `g.am.hPart` generator.

- `g.am.hPart` uses the generators `g.amOrStm` and `g.am.hPS` in conjunction with the extractors `e.all` and `e.f.mcs` to iteratively derive a homogeneous partition of an ensemble.

- `g.amOrStm` reads in trace record encoding sequences and outputs either an adjacency matrix or a transition matrix in canonical format.

- `g.autoCat` reads in a trace record encoding sequence and generates a sequence of ordered pairs representing points on a graph of categorical autocorrelation.

- `g.autoCor` reads in a trace record encoding sequence and generates a sequence of ordered pairs representing points on a graph of autocorrelation versus lag.

- `g.c1or2D` reads in a trace record encoding sequence and generates a sequence of ordered pairs representing points on a one-dimensional categorical visualization.

- `g.corrCat` reads in two ensemble members and generates a sequence of ordered pairs representing points on a categorical correlation.

- `g.cumCat` reads in a trace record encoding sequence and generates a sequence of ordered pairs representing points on a cumulative categorical periodogram.

- `g.ensStny` is a script that uses an extractor to read in an ensemble; for each member of the ensemble, it determines whether the hypothesis that the member is stationary with respect to either its timestamp distribution or its trace record encoding distribution is acceptable. This generator makes use of the `g.kw` generator to perform a Kruskal-Wallis rank sum test upon each of the input members. The results are written out in canonical format.

- `g.findPat` finds patterns in categorical data.

- `g.fitPath` reads in one or more ordered pair sequences in canonical format that define paths in two-dimensional Cartesian space; for each such path, it fits a straight line using linear least-squares data fitting. Its output consists of its input with each original sequence augmented with attributes that define the straight line fitting the curve and the quality of fit.

- `g.hist` generates from its input the number of bins and bin sizes for a histogram.

- `g.kw` uses the Kruskal-Wallis rank sum test to test the hypothesis that the values in each sequence in the canonical format input are drawn from the same distribution. Its output contains no sequences, only attributes that give the results of the test.

- `g.meg` reads in a sequence of trace record encodings and outputs in canonical format the points describing a mass evolution graph.

- `g.megRegressPart` uses the generators `g.meg` and `g.megSlope` to partition the sequences in its input into segments by using incremental regression.

- `g.megSlope` reads in the output of `g.meg` and outputs in canonical format the slopes of each of the paths of the input mass evolution graph.

- `g.megSlopePart` uses the generators `g.meg` and `g.megSlope` to partition the sequences in its input into segments within which the slope of the points contained remain the same within a certain threshold.

- `g.ots` reads in a sequence of ordered pairs in canonical format representing trace record encodings and occupancy times; it then outputs a matrix in canonical format containing various occupancy time statistics.

- `g.p.cat` reads in a trace record encoding sequence and generates a sequence of ordered pairs representing points on a categorical periodogram.

- `g.Pm` reads the output of `g.fitPath`; for each ordered pair sequence in the latter describing a group of points (x, y) to which a path has been fitted, `g.Pm` generates a new sequence with a single ordered pair (x', s) where x' is copied from the first ordered pair in the input sequence and s is the slope of the path. The new sequences collectively describe a probability mass function, hence this generator's name. This generator is used as part of the process of generating *mass evolution graph* models [18].

- `g.smooth` reads in a single ordered pair sequence from an ensemble member in canonical format and outputs a modified sequence representing a smoothing of the input. Two smoothing functions are available: exponentially-weighted moving average and uniformly-weighted moving average.

4.5.2 Transformers

A *transformer* is a CHITRA95 module that modifies an ensemble in some fashion, typically with the goal of either reducing or partitioning its data. All current transformers owe their existence to CHITRA95's model-building heritage; they have names of the form `x.<name>` and include the following

- `x.a.fil.ens` takes an ensemble and replaces one or more embedded runs of trace record encodings with an aggregate encoding. The runs are selected such that they satisfy some filtering criteria.

- `x.a.pat.ens` takes an ensemble and replaces one or more embedded runs of trace record encodings with an aggregate encoding. The runs are selected such that they form some specified pattern.

- `x.clip.ens` takes an ensemble and truncates one or more members to specified bounds.

- `x.p.ic` reads in an ensemble in canonical format. For each member of the ensemble, it converts the sequence of trace record encodings and accompanying timestamps, translates each trace record encoding into the corresponding trace record, and outputs a sequence of modified trace records containing selected modules only, along with the accompanying timestamps.

- `x.p.sr` takes an ensemble and scales to a specified range each of a specified set of trace record fields for each trace record in the ensemble. One of three scaling functions may be selected: truncation, multiplication by a scale factor, or taking the modulus.

- `x.part.ens` takes an ensemble and replaces it with an ensemble that is a partitioning of the original. It makes use of the `e.part.ens` extractor to perform the actual partitioning.

- `x.undo.ens` takes an ensemble and rescinds the most recent ensemble-wide modification performed upon it.

4.5.3 Synthesizers

A *synthesizer* is a CHITRA95 module that creates a synthetic trace in canonical format. The trace it creates is described as synthetic because it is generated according to writer-defined rules as opposed to being collected from a target. Such a trace might be used to drive a simulation model, in which case the synthesizer functions as a workload generator. Alternatively, a synthetic trace could be reused for validity testing on original traces.

CHITRA95 synthesizers have names of the form `s.<name>` and include the following:

- `s.clone` reads in an ensemble and creates a near duplicate, the only differences being in the creation date and member names.

- `s.fitPath` reads in one or more ordered pair sequences in canonical format describing paths in two-dimensional Cartesian space. Given the slope and intercept specified along with each of the input sequences, it generates a new sequence of ordered pairs that define points for a straight line with that slope and intercept.

- `s.gen` reads in the probability mass function output in canonical format by `g.Pm` and uses a uniform random number generator to generate a synthetic trace that satisfies the function.

- `s.sm.ens` reads in a transition matrix in canonical format and uses a linear congruential random number generator to generate a synthetic trace that exhibits the transition behavior described by the matrix.

4.6 The Presenter Layer

Above the generator/transformer/synthesizer layer lies the layer containing presenters. A *presenter* is a CHITRA95 module that takes the output from a lower-level module (e.g. an extractor or a generator) and formats it for presentation. The formatting involved can be as simple as adding headings to a text dump or as elaborate as bringing up a graph in a window. CHITRA95 provides the following presenters:

- `v.autoCat` creates a categorical autocorrelation graph of a given ensemble. It either displays the graph in a window on the user's screen or creates an output file in either Encapsulated PostScript (EPS) or Portable Bitmap (PBM) format.

- `v.autoCor` takes the 0-th member of a given ensemble and creates a graph showing autocorrelation as a function of lag for the function trace record encoding versus event number. The output is sent either to a window on the user's screen or to a file in EPS or PBM format. `v.multi.autoCor` is similar to `v.autoCor`, except that it creates an autocorrelation graph for up to three ensembles at the same time.

- `v.catPer` creates a categorical periodogram graph of a given ensemble. The output is sent either to a window on the user's screen or to a file in EPS or PBM format.

- `v.corrCat` graphs the categorical correlation between two selected members of a given ensemble. The output is sent either to a window on the user's screen or to a file in EPS or PBM format.

- `v.cumCat` creates a cumulative categorical periodogram of a given ensemble. The output is sent either to a window on the user's screen or to a file in EPS or PBM format.

- `v.findPat` creates a patterngram of a given set of files. The output is sent either to a window on the user's screen or to a file in EPS or PBM format.

- `v.fit0path` takes the output of `g.fitPath` and computes a fitted path using the synthesizer `s.fitPath`. The output is sent either to a window on the user's screen or to a file in EPS or PBM format. `v.fit1path` is similar, except that it passes the input sequences through `g.megSlope` and `g.megSlopePart` before synthesizing the fitted path.

- `v.gantt` takes an ensemble with a single member and graphs its trace record encodings as a function of either time or event number. The output is sent either to a window on the user's screen or to a file in EPS or PBM format.

- `v.hist` creates a histogram of the number of occurrences of each trace record encoding in a given ensemble. The output is sent either to a window on the user's screen or to a file in EPS or PBM format.

- `v.matrix` takes as input a matrix in canonical format and writes a nicely formatted version to standard output.

- `v.meg` plots a mass evolution graph of a given ensemble. The output is sent either to a window on the user's screen or to a file in EPS or PBM format.

- `v.ots` reads in a matrix of occupancy time statistics in canonical format generated by `g.ots` and writes a nicely formatted table of statistics to standard output.

- `v.pmf` takes the probability mass function generated by `g.Pm` and creates a nicely formatted version which it sends to standard output.

- `v.smooth` takes its output from `g.smooth` and creates a plot of the smoothed ensemble. The output is sent either to a window on the user's screen or to a file in EPS or PBM format.

- `v.stat.ens` computes and displays a nicely formatted table of occupancy time statistics for a given ensemble. It differs from `v.ots` in that `v.stat.ens` is a wrapper

that calls both `g.ots` and `v.ots` to create the table. The output is sent to standard output.

- `v.text` is a simple viewer that creates a nicely formatted text dump of either all or selected members of a given ensemble. The text dump lists event number, timestamp and trace record encoding for each sequence entry dumped. The output is sent to standard output.

There are two module subcategories closely related to presenters: modelers and testers. A *modeler* constructs an analytical model of an ensemble with the goal of distilling the essential behavior exhibited by the trace or traces that make up the ensemble in order to facilitate understanding. CHITRA95 comes with the following modelers:

- `m.meg.ens` generates a fitted mass evolution graph distribution and occupancy time statistics for a specified ensemble with one member.

- `m.sm.ens` generates a probability transition matrix and occupancy time statistics for a specified ensemble.

On the other hand, a *tester* is a CHITRA95 module that applies some statistical test to its input. Like a modeler, however, testers have closer ties to presenters than to generators because the output from a tester is already preformatted for presentation. CHITRA95 provides the tester `t.kw` which applies the Kruskal-Wallis rank sum test to its input.

4.7 The User Interface Layer

The final, topmost layer of the CHITRA95 logical design consists of user interfaces. In CHITRA95 terms, a *user interface* is an organizing program that incorporates lower-layer modules with the goal of providing a unified framework for performing trace analysis tasks. More specifically, a user interface is responsible for informing the user about available trace manipulation/analysis tasks, and for providing the user with the means to both select and execute those tasks. How these responsibilities are fulfilled without committing to a specific

human-computer interaction model (e.g. command-line-based, graphical-interface-based) is one of CHITRA95's major innovations. CHITRA95 user interfaces decouple themselves from specific human-computer interaction models by stressing content over appearance. Configuration files are used to describe each module and its place within the interface; these files specify what arguments are required, what options may be selected and what flags may be set. They also describe the nature of the output produced by the modules. The configuration files are then processed using one of two approaches:

- By means of a root display, CHITRA95 presents the user with a list of possible actions. Once selected, the corresponding module dynamically creates the necessary portion of the user interface (e.g., any required dialogues or output windows).
- A separate CHITRA95 utility processes the configuration files and creates a static shell with which the user may then interact.

The first approach has the advantage of flexibility because the configuration of the interface conforms with available modules; on the other hand, the approach requires a scripting language capable of producing a user interface, such as Tcl/Tk [16]. The second approach is simpler because all the information required to build the interface is derived once, prior to running CHITRA95; however, the user needs to explicitly rebuild the interface whenever a new module is added. Despite the drawbacks attendant with either approach, extensibility is still achieved with little interface-related programming costs precisely because the specification given by a configuration file is at such a high level.

Another innovation is how the interface is made extensible without necessarily requiring programming on the part of the person making the extension. Each module in CHITRA95 conforms to the interaction model show in Figure 4.2. In this model, a conforming module is invoked via a command-line, the syntax of which is specified in the module's configuration file. The module reads from standard input and writes to standard output, both of which are piped in from and out to the user interface. The user's interaction with the interface is translated into the appropriate command line; the module is invoked and its output

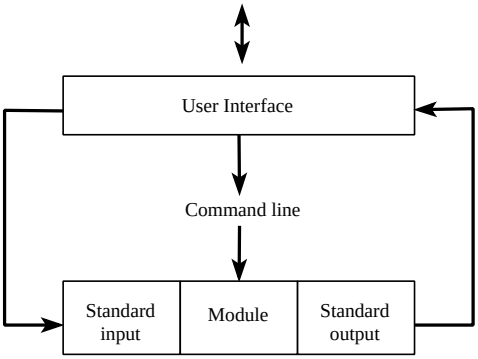

Figure 4.2: CHITRA95 Module Interaction Model

processed into presentable form by the user interface. New modules added by third parties are automatically supported by the interface architecture, as long as the modules conform to this interaction model and the appropriate configuration files are written.

Research is underway on using World Wide Web browsers as a means of achieving user-interface portability; Figure 4.3 shows a prototype of the interface to the "WebScope" tool [6]. To a certain degree, a World Wide Web browser shares with a CHITRA95 interface the greater attention paid to content as opposed to mere appearance. A forms-based interface, for example, would be easily accessible and, as a bonus, would require no graphical user interface programming at all. On the other hand, the intense competition in establishing browser standards brings about the introduction of non-standard browser features which might prove tempting to support, which in turn would diminish the interface's portability.

4.8 Module Verification

In CHITRA95, the combination of small, modular components and a text-based data interchange format makes it simple to add regression testing. Each module group provides an executable script that in turn runs a test script for each module in the group. The test script runs the module against one or more known sample inputs; the module output is then compared against another known, pre-validated sample. Any differences are then sent to

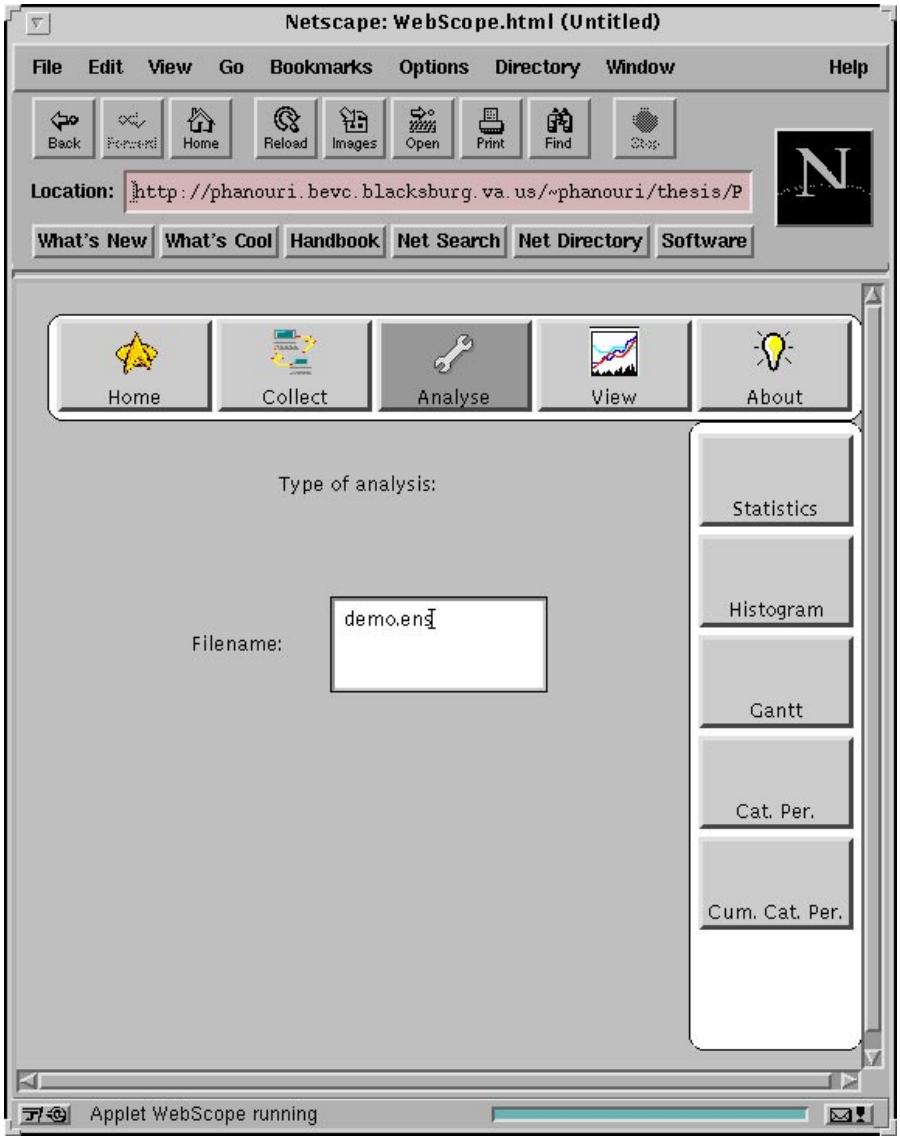

Figure 4.3: *WebScope* Prototype User Interface

standard output, at which point the person running the test script can determine whether the differences are benign or indicative of faulty module operation. The verification process is thus reduced to simple visual inspection of the test script output.

In addition to providing test scripts for each module, CHITRA95 also provides a more comprehensive, integrated test based on the CHITRA93 test suite. This test is composed of a series of complex operations involving several modules from different module groups. Again, the inputs are known and the sample output or outputs are pre-validated; verification consists of inspecting the test output for differences from the sample output or outputs.

4.9 Summary

CHITRA95 is designed with five primary goals in mind: extensibility, reusability, portability, reliability and maintainability. The first goal, extensibility, is promoted by a modular, layered design that makes use of a common, text-based data interchange format, permitting new modules to be written in the user's language of choice. The separation of computation and interface allows the module writer to concentrate on one of the two. Reusability is automatically promoted by the adherence to modularity and layering, in addition to the use of the aforementioned data interchange format. The design is portable both from an operating system/environment standpoint and from a human-computer interaction standpoint. At the same time, the functional separation along strict lines promotes reliability by limiting the effects of failures in modules under development and by permitting module verification to be performed on an incremental basis. Finally, maintainability is promoted by the tendency towards smaller, easier to understand programs and by the clear separation of duties between modules.

Chapter 5

THE IMPLEMENTATION OF Chitra95

The design presented in Chapter 4 gives the logical structure of CHITRA95. In this chapter, we describe how each of the layers of the logical design are actually implemented in terms of source code.

5.1 Implementation Choices

CHITRA95 is written primarily in C++ using a methodology called *literate programming* [11]. The choice of C++ was influenced primarily with our desire to increase code reuse and apply object-oriented principles to the CHITRA95 design. Also, the existence of pre-built class libraries, such as `libg++` [12], simplified the initial implementation by sparing us the effort of writing our own data structure library. CHITRA95 does not make use of the new Standard Template Library (STL) [15] because much of the CHITRA95 was written before STL was first released.

The use of literate programming was motivated by our observation that the writing of code almost always preceded the writing of corresponding documentation, decreasing the probability that the latter would actually get written. Literate programming is a discipline that requires programs to be written as if they are to be read and understood by disinterested parties. Often, it involves the writing of extensive documentation *at the same time* as code and therefore makes use of tools that generate both formatted documentation and compilable source code from a single source document. The tool we have employed in writing most of CHITRA95's source documents is called *FunnelWeb*.

5.2 Organization

The source code for CHITRA95 is organized into module groups that roughly parallel the demarcation lines between the logical layers described in §4.1. These module groups are as follows:

common: This module group contains CHITRA95 data types, global symbols, system-dependent definitions and a number of generic helper functions. It also contains the class definitions that form the basic building blocks from which much of CHITRA95 is built: the abstract data type classes; the `Args` argument parser class; the `bfstream` binary `fstream` class; the `CFStream` canonical format stream class; the `ListSpec` list specification parser class; the `MapHash` mapped hash table class; the `Object` root class; and a simple implementation of the ubiquitous `String` class.

ensemble: This module group contains the definitions used in the ensemble and ensemble member representations, namely class `Ensemble`, class `Member`, class `StateMap` and class `Extents`. It also contains definitions for various helper classes, namely `MatchSummary`, `RunFinder` and `SetFinder`, and a library of ensemble-related helper functions.

assistants: This module group contains programs for assisting in assorted tasks, hence the module group's name. Most of the programs are concerned with translating between various file formats. The inserters `a.t.cf2ens` and `a.import.ens` are in this module group. Also in this module group are `a.map.s2m`, a program for mapping trace records contained in a `CFStream` to trace record encodings and `a.mod.set`, a program for performing simple set operations (written primarily for use by the generator `g.am.hPart`).

extractors: This module group contain the extractors described in §4.4.

generators, synthesizers, transformers: These three module groups contain the generators, synthesizers and transformers described in §4.5.

modelers, testers, viewers: These three module groups contain the viewers, modelers and testers described in §4.6.

testSuite: This module group contains a collection of tests adapted from the CHITRA93 test suite.

5.3 Building Blocks

At the lowest level, CHITRA95 is built upon a number of custom building blocks which serve to isolate environment dependencies as well as provide functionality not otherwise available from either the C or C++ standard libraries. These building blocks include the data types, class definitions and functions collectively known as the **common** module group.

5.3.1 Basic Data Types

The data maintained by various CHITRA95 is defined in terms of several basic data types, namely `Integer`, a signed integer type at least 32 bits long; `UInteger`, an unsigned integer type at least 32 bits long; `Real`, a floating-point type equivalent to or better than IEEE double-precision; `Byte`, an unsigned 8-bit integer type; and `Time`, an unsigned integer type at least 32 bits long. CHITRA95 also defines `Bool`, a Boolean data type and the constants `TRUE` and `FALSE`. Finally, CHITRA95 defines an unsigned integer type called `MState` (otherwise known as a trace record encoding).

The basic data type definitions are contained in the file *types.fw*.

5.3.2 Abstract Data Types

CHITRA95 defines a number of abstract data types (ADTs) written as C++ class templates. The templates are patterned after `libg++` instead of the C++ Standard Template Library for historical reasons. Access into an ADT instance is provided via an `Iterator` data type; dereferencing an `Iterator` returns the data item it points to. Any ADT instance may be written to or read from an open `bfstream` instance.

The abstract data types defined by CHITRA95 are:

List: A classic doubly-linked list template class, `List` provides the following methods: `Clear`, for initializing the list; `Length`, which returns the number of items in the list; `InsertBefore` and `InsertAfter`, both of which insert a new item before (or respectively, after) a specified position; `Delete`, which removes the item at a specified position; `Prepend`, which adds an item at the front of the list; `Append`, which adds an item at the end of the list; `Contains`, which determines whether a specified item already exists in the list; `Find`, which returns an `Iterator` pointing to the first occurrence of a specified item, if any; and `IsEmpty`, which returns TRUE if and only if the list contains no items. Also provided are four traversal methods that return `Iterator`s, namely `First`, `Last`, `Prev` and `Next`.

Tree: A simple unbalanced binary-tree implementation, `Tree` provides the following methods: `Clear`, for initializing the tree; `Insert`, which adds an item to the tree; `Delete`, which deletes a specified item from the tree; `Contains`, which determines whether a specified item already exists in the tree; `IsEmpty`, which returns TRUE if and only if the tree contains no items; `NumItems`, which returns the number of items in the tree, and `Find`, which returns an `Iterator` pointing to the first occurrence of a specified item, if any. A full set of traversal methods is also provided.

Stack: A classical push-down stack implementation, `Stack` provides the following methods: `Clear`, for initializing the stack; `Depth`, which returns the depth of the stack (or equivalently, the number of items in the stack); `Push`, which adds an item to the top of the stack; `Pop`, which returns the item at the top of the stack, deleting the item at the same time; and `IsEmpty`, which returns TRUE if and only if the stack contains no items. Traversal methods that return `Iterator`s are also provided; these are called `Top`, `Bottom`, `Down` and `Up`.

HashTable: An open-bucket hash table implementation, `HashTable` provides the following methods: `InitBuckets`, for initializing the number of buckets to a specified value;

`Clear`, for initializing the hash table; `Insert`, which adds an item to the hash table; `Delete`, which deletes a specified item from the hash table; `Contains`, which determines whether an item already exists in the hash table; `IsEmpty`, which returns TRUE if and only if the hash table contains no items; `NumItems`, which returns the number of items contained by the hash table; and `Find`, which returns an `Iterator` pointing to the first occurrence of the specified item, if any. Also provided are four traversal methods, namely `First`, `Last`, `Prev` and `Next`. Unless a `HashTable` is instantiated with a custom hash function, the default hash function is used; this function hashes all items to the bucket 0.

Set: A simple set implementation, `Set` provides the following methods: `Clear`, which initializes the set; `Add`, which comes in two versions, one for adding a single item and another for adding all the items contained in a `Set` instance of the same type; `Delete`, which also comes in two versions, one for deleting a specific item and another for deleting the first instance of a sample item; `NumElements`, which returns the number of elements in the set; `Contains`, which determines whether a specified item already is an element of the set; `Find`, which returns an `Iterator` pointing to the first occurrence of the specified item, if any; and `IsEmpty`, which returns TRUE if and only if the set contains no elements. As with the other classes described above, `Set` provides a full set of traversal methods, namely `First`, `Last`, `Prev` and `Next`.

Array: A dynamically-sized array that automatically accommodates references made to indices beyond the currently known end-of-array index, `Array` overloads the `[]` operator to provide transparent access to arbitrary array elements. `Array` also provides the following methods: `MaxIndex`, which returns the currently defined end-of-array index; two versions of `Find`, one which returns the index of the element that exactly matches a specified sample and another which returns the index of the element that satisfies a specified criterion; and `Sort`, which sorts a specified portion of the array in ascending order.

SparseMatrix: A NxN sparse matrix implementation, `SparseMatrix` provides the following methods: `Clear`, which initializes the matrix; `RowSum`, which returns the sum of the elements in row $0 \leq i < N$; `Print`, which prints the matrix to standard output; `Dimension`, which returns the dimension N of the matrix; `Put`, which sets element (i, j) to a specific value; `Get`, which returns element (i, j); and `Inc`, which increments element (i, j).

All abstract data type class definitions may be found in the file *adt.fw*.

5.3.3 Base Classes

The persistent data types in CHITRA95 (i.e. the classes that compose the ensemble and ensemble member representations) are derived from class `PersistentObject`, which in turn is derived from class `Object`. Class `Object` is used for maintaining status information relating to a specific class instance. Towards this end, it provides the following methods: `OK`, which returns TRUE if and only if the current instance status is non-zero; `Status`, which returns the actual current instance status value; `SetError`, which sets the instance status to some value; `ClearError`, which resets the instance status to zero; and `ErrorMessage`, which returns an explanatory string corresponding to the current instance status.

The `PersistentObject` class adds a few methods for managing persistent storage and instance version. These methods include `Open`, which associates persistent storage with the object instance, loading the latter with data from the former if necessary; `Close`, which disassociates persistent storage from the object instance after saving any instance data; `Handle`, which returns the name associated with the persistent storage; `Version`, which returns the current version of the object instance; `StartNewVersion`, which prepares the object instance for a new version (which may involve manipulating data in persistent storage); and `Undo`, which rescinds all changes made since a specified number of versions were started.

The source code for the `Object` and `PersistentObject` class definitions appears in the file *object.fw*.

5.3.4 The `CFStream` Class

The `CFStream` class implements the CHITRA95 canonical format as an ASCII data stream similar in concept to a C++ `iostream`. The data stream is a record-oriented format, the general structure of which is depicted in Figure 5.1. The stream contains the data for N

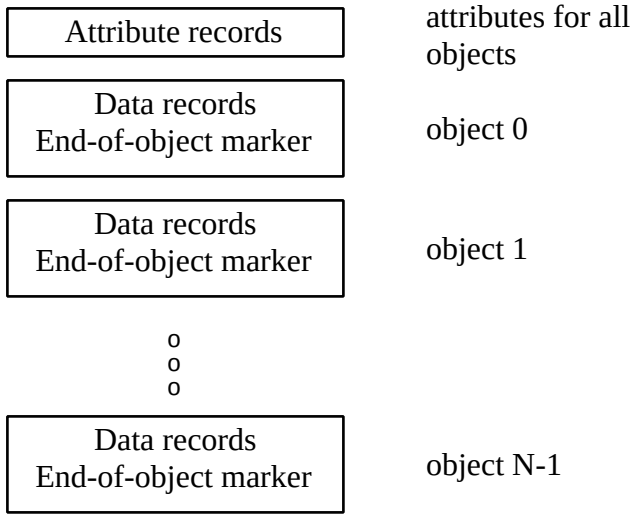

Figure 5.1: General structure of a `CFStream`

distinct objects, each of which is annotated by one or more *attribute records*. An attribute record consists of an *attribute name* prefixed by a `'#'` character; an optional *object number*, separated from the attribute name by a `'.'` character, which identifies the object to which the attribute applies; and the *attribute value*. Attribute records without associated object numbers are called *global attributes*; they either apply to all objects in the stream or describe some property of the stream itself (e.g. as in the case of *receipt records* which track the stream's history in terms of the `CFStream`-aware programs through which the stream has passed).

The data for each object is organized into *tuples*. In general, all tuples belonging to the same object are of uniform length. The data for each tuple is gathered into a *data record* which is simply a line delimited by whitespace and terminated by a newline character.

A special type of record called a *control record* is used to keep track of record and object boundaries[1] Finally, *comment records* (which are prefixed with the string ``##'') may appear anywhere in the stream; these serve primarily as an aid to human readers and are ignored during parsing.

Input and output of attribute data are performed by means of `GetAttribute` and `PutAttribute` calls. The `CheckAttribute` method may be used to determine whether a specified attribute has a specified value. The `GetGlobalHeader` method may be used to determine the number of objects in the stream. An output method named `PutGlobalHeader` may be used to emit the same information to an output stream. The `CountAttribute` method may be used to count the number of objects in the stream containing a particular attribute with a specific value. Input and output of data records are performed on a per-field basis with the `>>` and `<<` operators. The `AddToReceipt` method may be used to add a new receipt record to the set already contained in the incoming stream data. The `PutEndOfObjectMarker` and `PutComment` methods may be used to insert an end-of-object control record and a comment record, respectively, at the current position in the stream.

The current input pointer may be advanced by calling either the `AdvanceToNextObject` method or the `SkipToEndOfObject` method. The end-of-object and end-of-stream conditions may be tested by calling the Boolean methods `EndOfObject` and `EndOfStream`, respectively.

Although a `CFStream` may contain objects of any type, the `CFStream`s most often generated by the provided CHITRA95 modules are *sequences* and *matrices*. The `CFStream` class interface therefore provides a number of convenience functions related to these two object types. The `GetSequenceHeader` and `GetMatrixHeader` methods retrieve the values of attributes of general interest related to a specific object. The `GetMStateInfo` method retrieves statistics concerning trace record encoding statistics. `CFStream` provides matching output methods named `PutSequenceHeader`, `PutMatrixHeader` and `PutMStateInfo`,

[1]The current ASCII-only implementation requires end-of-object control records only.

respectively.

The file *cfstream.fw* contains the source code for the `CFStream` class definition. It also contains the definitions of a large number of macros corresponding to attribute names and attribute values.

5.3.5 Other Classes and Functions

CHITRA95 also defines a number of smaller, though no less important classes in its private library, namely `Args`, `bfstream`, `ListSpec`, `MapHash` and `String`.

The `Args` class implements a command-line argument parser that takes a logical description of a valid argument line and the set of arguments passed in through the `argc/argv` mechanism available to a C++ program. The argument line is described in terms of three possible argument types: *positional parameters*; a *switches*; and *named options*. A switch is a Boolean argument prepended by a `'-'` character (e.g., `"-debug"`). A named option is like a switch except that it does not denote a Boolean argument and may be followed by an option value (e.g. `"-member <member-list>"`, where `<member-list>` is a list of one or more member IDs). All other arguments are positional parameters; they are individually numbered, starting at 1 (the full pathname to the program itself is argument 0). The `Args` class provides the `DescribeSwitch` method for describing switches and named options and the `DescribeParam` method to describe positional parameters; matching methods named `QuerySwitch` and `QueryParam` are also provided for retrieving argument values. The source code for the `Args` class appears in the file *args.fw*.

The `bfstream` class is a binary file input and file output stream class derived from class `fstream`. It overrides all input and output operators such that they perform raw binary reads and writes of the specified data, as opposed to the normal `fstream` behavior of translating from and to ASCII upon input and output. In addition, class `bfstream` provides two new methods named `lock` and `unlock` that permit exclusive access to the associated file to be turned on and off on command, and also overrides the `close` method to ensure that the associated file is unlocked prior to being released. The source code for

the `bfstream` class appears in the file *bfstream.fw*.

The `ListSpec` class takes a string containing a *list specification* (e.g. `"1,2,3-7"`) and parses it so that the list's individual elements can be accessed in sequence using methods named `First` and `Next`. As indicated in the example, the list may name elements explicitly or may indirectly specify one or more elements by means of ranges. The source code for the `ListSpec` class appears in the file *listspec.fw*.

The `MapHash` class is another hash table class; it is used by the generators `g.autoCat` and `g.corrCat`. The source code for the `MapHash` class appears in the file *maphash.fw*.

The `String` class is an implementation of a character string class, hence its name. Most C++ libraries that predate the Standard Template Library do not provide a string class, hence the decision to include one in the CHITRA95 library. The source code for the `String` class appears in the file *string.fw*.

Finally, CHITRA95 includes a number of helper functions in its private library. These include `startup`, a standard program preamble routine; `crash`, which outputs specified messages to standard error before calling `exit(2)`; `Terminate`, which emits an end-of-object marker to a specified `CFStream` instance and exits; `StrToModeT`, which converts a file permissions mode string to a `mode_t`; `MakeDirectory`, which creates a named directory with a specified set of permissions; `IsAccessible`, which checks a file or directory for specified access privileges; `LockFD` and `UnlockFD`, which lock and unlock, respectively, a given file descriptor; `Pow`, which raises a `UInteger` to some power; `PrintDashes`, which prints a line of dashes of a specified width to standard output; `ReadString` and `WriteString`, which read and write, respectively, possibly quote-delimited strings from and to a specified stream; and a full set of relational operators templates. The source code for all these functions appears in the file *helpers.fw*.

5.4 The Ensemble Representation

In CHITRA95, the data that comprises an ensemble is maintained in an instance of class `Ensemble`. Each such instance keeps together the data for one or more traces and provides methods for accessing both trace data and commonly-used statistics about each trace and about the ensemble as a whole. The data for each trace in the ensemble constitutes an ensemble member and is kept in an instance of class `Member`. For each ensemble, a single instance of class `StateMap` maintains a record of all mappings between trace records and trace record encodings for all ensemble members. The *undo* operation is made possible at the ensemble member level by employing in each ensemble member an instance of class `Extents`.

In order to support the data searching requirements of the transformers `x.a.pat.ens` and `x.a.fil.ens`, CHITRA95 provides a helper class named `Finder`; instances of this class are associated with specific ensemble instances and work directly with ensemble data instead of working through the `CFStream` interface. CHITRA95 also provides a number of helper functions, some of which take an ensemble instance as part of their arguments; these functions include `ParseMemberIDSpec`, which parses a specification of a list of member id numbers; `ParseModelStateSpec`, which parses a specification of a list of model states, otherwise known as trace record encodings; `TimestampToCursor`, which converts a timestamp into an index into an ensemble member; `BuildMSAggregate`, which constructs the definition of an aggregate model state (i.e., an aggregate trace record encoding); and `RangeCompress`, which collapses a list of explicitly specified numbers into one or more ranges, if possible (e.g., the list ‘‘`1,2,3,4,7,10,11,12`’’ is collapsed into ‘‘`1-4,7,10-12`’’).

5.4.1 The `Member` Class

The matched timestamp and trace record encoding sequences that make up an ensemble member are represented by a series of records in persistent storage (i.e. a file on disk). Each record consists of a `Time` instance and an `MState` instance. Records are numbered in

sequence according to their physical offset relative to the beginning of the file.

Typically, an ensemble member is built up by a series of *append* operations. In this case, the physical sequence of records in the file directly corresponds to the logical timestamp and trace record encoding sequences. The length of the physical sequence can and often does reach into the tens or even hundreds of thousands, which clearly makes it infeasible to implement a logical operation, say insertion of a new record at position 0, naively in terms of its physical equivalent, in this case amounting to an expensive move of all but one of the physical records by one record position. Additionally, the potential for a subsequent *undo* operation underscores the need for an efficient and easily reversible method of performing all operations that modify the member.

The method employed by CHITRA95 is to describe the physical record sequence in terms of extents. An *extent* is a 3-tuple (P, O_S, O_E) where P, O_S and O_E denote the logical position, a starting physical offset and an ending physical offset, respectively, of a contiguous sequence of physical records within the ensemble member. Each reference to a logical position is translated into a physical equivalent which in turn is used by lower-level file input/output routines. For example, an ensemble member of length L that has been built up solely by a series of *append* operations may be described by the single extent $(0, 0, L-1)$, as shown in Figure 5.2. In this figure, the boxes represent physical records in persistent

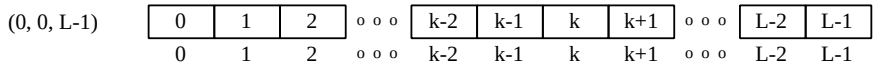

Figure 5.2: A freshly-created ensemble member

storage. Each record is internally labeled with its physical offset; the corresponding logical offset appears immediately underneath.

The insertion of a new record into logical position 0 can then be performed by physically appending the new record to the record sequence (i.e., at position L) and describing the member with the *extent sequence* $(0, L, L)$, $(1, 0, L-1)$; Figure 5.3 shows the resulting record sequence; note that the logical offsets have shifted by one position.

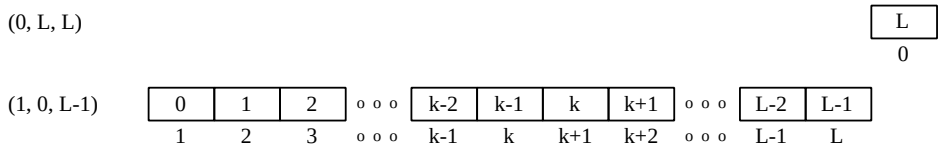

Figure 5.3: The ensemble member after inserting a new record at position 0

The deletion of a record may split one of the extents in the extent sequence; in the current example, deleting record k where $0 < k < L$ yields the extent sequence $(0, L, L)$, $(1, 0, k-2)$, $(k, k, L-1)$; note, however, that the physical record corresponding to logical position k need not be deleted from the member, and that the *undo* operation needs simply to undo the modification made to the extent sequence, not to the physical record sequence itself. Figure 5.4 shows the record sequence after the deletion. Note that while the record

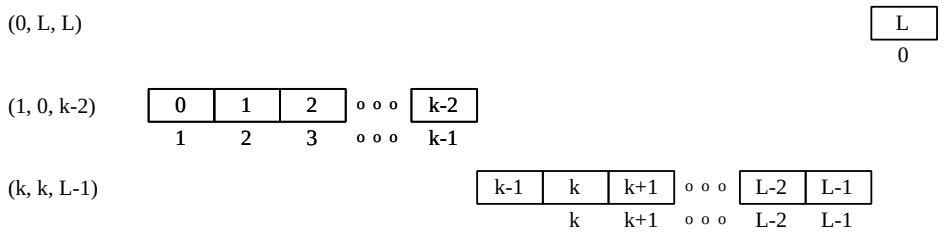

Figure 5.4: The ensemble member after deleting the record at position k

at physical offset $k-1$ is still physically a part of the record sequence, it has been logically excised.

The `Extents` class, an instance of which is carried by each `Member` instance, implements an extent sequence with undoable insert, delete and append operations by maintaining a *stack* of extent sequences instead of just one. The *start-new-version* operation pushes a copy of the top of the stack (i.e. the current extent sequence) onto the stack; the *undo* operation then merely pops the stack as many times as there are versions to be rescinded. All `Member` methods that modify the record sequence also modify the extent sequence on the top of the stack accordingly. As an optimization, no logical-to-physical translation is

performed when the `Extents` instance is at version 0, nor is such a translation required.

The `Member` class also maintains stacks for two attributes that need to be undoable, namely the starting time of the member (subject to modification in case one or more records are deleted from the beginning of the member) and the length of the member (which is cached instead of being computed every time it is needed).

The persistent storage for each `Member` instance consists of a directory named after the member's handle; a file containing the sequence of trace records; a file containing the data for the `Extents` instance; and a file containing additional attributes connected with the trace.

In addition to implementations of the logical operations described in §4.2, the `Member` class adds the following set of methods: `SetAttributes`, which sets the values of the member attributes *source trace name*, *timestamp type*, *starting time* and *ending time*; access functions for each of the aforementioned member attributes; and methods required by the `PersistentObject` class from which class `Member` is derived.

The source code for and detailed descriptions of the `Member` and `Extents` classes appear in the files *enmbr.fw* and *enext.fw*, respectively.

5.4.2 The `StateMap` Class

The `StateMap` class maintains a set of tables for one or more mappings of trace records to trace record encodings. Three types of mapping are supported, the first of which is a direct translation from a trace record to a trace record encoding. The second mapping type is a translation from one trace record encoding to another encoding. The third mapping type defines a translation from either a set of trace record encodings or a sequence of trace record encodings. Overloaded `Map` methods are provided for each mapping type; at the same time, complementary `Unmap` methods are also defined.

Version management is made possible by tagging each mapping entry with the version number that is outstanding at the time that the entry is made. Starting a new version consists simply of incrementing the version number; new mapping entries are then tagged

with the higher version number. The *undo* operation decrements the version number, after which it goes through the mapping tables and invalidates each entry tagged with a version number higher than the "new" version number.

In addition to maintaining a set of mapping tables, `StateMap` also maintains the definitions of the fields (not including the timestamp field, if present) which comprise the *prototypical trace record* used for each member of an ensemble. The method `DefineComponent` permits the prototypical trace record to be defined one field at a time; once completed, field values may be individually associated with descriptive names with the `DefineComponentValue` method, variants of which exist for each CHITRA95 basic data type. The *start-new-version* and *undo* operations do not affect the prototypical trace record definition because the definition is not affected by any other supported ensemble or ensemble member operation.

The source code and detailed description of the `StateMap` class appear in the file *enmap.fw*.

5.4.3 The `Ensemble` Class

The `Ensemble` class presents an interface to one or more `Member` instances and provides facilities, courtesy of an embedded `StateMap` instance, for maintaining the ensemble's prototypical trace record definition and performing translations between trace records and trace record encodings.

After a new member is added via the `AddMember` method and its identifying ID number determined, the member may be manipulated via a set of methods similar to that provided by the `Member` class, the difference being that the ID number must be specified in order to distinguish among the ensemble members. `Ensemble` maintains a simple list of member handles; the persistent storage for each member is kept within the directory created and maintained by `Ensemble`. In addition to the ensemble member directories, `Ensemble` maintains a file containing the data for the `StateMap` instance and another file containing additional attributes connected with the ensemble.

The `Ensemble` class also provides methods for computing ensemble-wide aggregate and

statistical information, such as extracting the set of trace records which occur in at least one ensemble member and calculating the total length of the ensemble. These methods typically call upon counterparts at the ensemble member and then merge the individual results as appropriate.

Because the `Ensemble` class is largely an interface for the `Member` and `StateMap` classes, the implementations of own *start-new-version* and *undo* operations simply invoke the corresponding methods of each `Member` instance and the single `StateMap` instance. They also increment and decrement, respectively, the ensemble's version number.

The source code and detailed description of the `Ensemble` class appear in the file *enens.fw*.

5.5 The Inserter/Extractor Layer

CHITRA95 provides a generic inserter named `a.t.cf2ens` which takes a canonical format data stream containing an ensemble and converts it into persistent form (i.e. an instance of `Ensemble`). `a.t.cf2ens` expects timestamped data, either in the form of unencoded trace records or of trace record encodings; the timestamps may either be absolute timestamps or occupancy times. The data stream may also contain a description of the ensemble's prototypical trace record, as well as a list of trace records that actually appear in the data stream; these are recorded in the ensemble by `a.t.cf2ens`. If the incoming data stream contains unencoded trace records, `a.t.cf2ens` takes care of encoding them to suit the needs of the internal member representation.

However, before a trace file can be added to an ensemble, it must first be converted into canonical format. Trace files come in several formats of their own, including comma-delimited or tab-delimited text; attribute-tagged text (e.g. Pablo's SDDF format); and CHITRA93's CSL format. All trace files must be converted into CHITRA95's canonical format before they can be processed by `a.t.cf2ens`. Such conversion is the task of *translators*, of which CHITRA95 provides the following:

a.t.cern2csl: Translates the Common Log File format generated by HTTP servers into CHITRA93's CSL format.

a.t.cf2gnu: Translates a canonical format file containing data points into an input file in the format expected by the graphing program `gnuplot`.

a.t.csl2cf: Translates a CHITRA93 CSL format file into CHITRA95's canonical format.

a.t.data2cf: Translates one or more data files in various text-oriented formats into CHITRA95's canonical format.

a.map.s2m: Replaces unencoded trace records in the canonical format input with trace record encodings; all pertinent information is converted as needed, and the results are output in canonical format.

The programs `a.o.csl` and `a.import.ens` are actually scripts that invoke translators to convert one or more trace files into canonical format. The converted traces are then fed to `a.t.cf2ens` for actual conversion into the persistent form.

Once the ensemble data has been converted into the persistent form, it may be extracted. The most commonly used extractor is named `e.all`. Typically, `e.all` is used to simply dump the entire contents of the ensemble in canonical format. However, `e.all` provides a variety of options for selecting which members to include in the output; for including or suppressing trace record, trace record encoding and/or timestamp information; for including or suppressing the trace record to trace record encoding table; and for selecting the type of timestamp information to output, regardless of whether the original ensemble timestamps consist of absolute timestamps or occupancy times. `e.all` maps an `Ensemble` instance to an ensemble in persistent form and makes use of various ensemble-level and member-level calls to generate its data, which is then sent through an instance of `CFStream` to produce output in canonical format.

The remaining extractors, namely `e.f.mcs`, `e.is` and `e.part.ens` are specializations of `e.all`, in the sense that they are structurally identical to `e.all`.

Source code, detailed descriptions and manual pages for all modules described in this section may be found in the **assistants** and **extractors** module groups.

5.6 The Generator/Transformer/Synthesizer Layer

The general form of a generator is a program that takes its input in canonical format, performs some computation upon the data contained, optionally adds or deletes one or more attributes and outputs its results in canonical format. All generators supplied with CHITRA95 are *intermediate* modules, in the sense that their output is either always sent to another module in a pipeline, possibly another generator, or to a higher-level presenter capable of formatting the output in a human-readable fashion.

On the other hand, the transformers provided with CHITRA95 work with an instance of the `Ensemble` class, that is to say, directly with an ensemble in persistent form. There is no prohibition against working with input and output in canonical format; however, since transformers typically edit one or more members of the ensemble, directly manipulating the ensemble's persistent form is more efficient. Although this is a clear violation of the separation between layers that characterizes the design, it was decided in the beginning that the need for efficiency outweighed the drawbacks; ironically, the evolution of the CHITRA user base's research interests has steadily led away from the creation of trace models (an activity heavily dependent upon the use of transformers) towards research into novel methods of visualizing trace data without necessarily applying the data reduction or data partitioning for which the transformers were originally written.

5.7 The Presenter Layer

All presenters provided by CHITRA95 take an ensemble in canonical format and produce one or more types of output: formatted text, a display window using the currently-running graphical windowing system, or an image written either in Encapsulated PostScript or one of the other binary image formats such as Portable Bitmap (PBM).

Regardless of output type, each presenter has a basic form: given the name of an ensemble, it invokes an extractor to obtain the required ensemble data. The data may then be formatted by the presenter itself if no computation needs to be performed with the ensemble data; this is the approach taken by the presenter `v.text`. More commonly, however, the data from the extractor is first piped to one or more generators and possibly other subsidiary presenters, the output of which is then formatted by the primary presenter, as in the case of `v.stat.ens` which uses `g.ots` and `v.ots` to create a table of occupancy time statistics.

Instead of generating formatted text, a presenter may use a helper program to display its data in a window. In this case, the presenter may use another helper to massage the data into a format suitable for input to the helper program. For example, the presenter `v.hist` uses the `gnuplot` program to create a histogram in a window of its own. The presenter `v.meg` not only makes use of `gnuplot`; it also uses the helper `a.t.cf2gnu` to convert a canonical format stream into a set of points plottable by `gnuplot`.

The third output option, that of generating a binary image or Encapsulated PostScript file, is a function of the helper program used by the presenter. Specifically, the `gnuplot` program can output either a binary image file or an Encapsulated PostScript file; this useful feature is taken advantage of by several presenters which make the feature available via command-line switches.

5.8 The Interface Layer

Of the two approaches to constructing an interface described in §4.7, CHITRA95 provides a skeletal implementation of the first approach (i.e., dynamic creation of interface components on an as-needed basis) centered around the `CfgArgs` class. `CfgArgs` is a wrapper class that parses a configuration file and builds a description of a valid command line by making the corresponding calls to an instance of the `Args` class. `CfgArgs` therefore provides a command-line user interface. By deriving a new class from `CfgArgs` and overriding the

methods `GetSwitch`, `GetOption` and `GetParameter`, a script for generating a suitable user interface may be generated instead of building a command-line description.

5.9 Module Verification

As outlined in §4.8, CHITRA95 makes use of a number of module-specific scripts to perform module verification. Each of the module groups named in §5.2 contains a subdirectory named **tests** containing the test scripts, input files in canonical format and corresponding output files. The test script for each module (named `RunTestSuite.<module>`) executes the module with one or more input files; the expected output is then compared against what the module actually produces after all irrelevant lines of output such as `CFStream` comment and receipt records have been filtered out by using the `grep` program. The differences, determined by using the `diff` program, are then sent to standard output, making it easy to identify module failures by inspection. All the tests for a module group may be performed by running the `RunTestSuite` script in the corresponding **tests** subdirectory. A global `RunTestSuite` script is provided for performing the tests for all module groups.

5.10 Summary

Written in C++ using a methodology called literate programming, CHITRA95 is organized into module groups that roughly mirror CHITRA95's logical structure. At the lowest level, CHITRA95 defines its own set of basic data types and a number of abstract data types as well. These, with the addition of the `Object` and `PersistentObject` base classes, the `CFStream` class that implements CHITRA95's canonical data format and other smaller classes form the core CHITRA95 library.

Ensemble data is maintained in an instance of class Ensemble, itself an interface to one or more instances of class `Member`, each of which maintains the data for a specific trace and an instance of class `StateMap` which records all known trace records and their corresponding encodings. These classes provide editing methods, the effects of which may

be undone if so desired. The `Ensemble` class is used primarily by inserters and extractors; transformers, although logically residing at a higher layer, use the `Ensemble` class as well to boost performance.

CHITRA95 provides a wide array of modules at the generator/transformer and presenter layers, a fact that speaks to CHITRA95's ability to support a user base of varying programming skill and research interests. By promoting the use of scripts that test for compliance with known output, CHITRA95 allows the module writer to design and implement his or her own tests without having to involve others who may not necessarily have experience in the same knowledge domain.

Chapter 6

LESSONS LEARNED

The design of CHITRA95 was driven largely by the lessons learned from the preceding generations of the tool (i.e., CHITRA92 and CHITRA93). However, once the initial design had been completed and members of the CHITRA research group started using the tool, a dialectical tension between intended and actual usage of the tool caused an evolution in the tool's design and implementation. In this section, we discuss the various lessons gained from that experience.

6.1 A modular architecture is flexible but slow.

Up until CHITRA93, each succeeding generation of CHITRA added a host of new features to a single, monolithic program with an integrated graphical user interface. This meant that each modification required the programmer to not only understand what the feature was about, but also to have a good grasp of the graphical windowing system the tool used. Each generation took an increasing amount of time to test and debug, largely because of the increase in program size and complexity.

All this was in stark contrast with our experience with the modular design of CHITRA95. A single programmer was responsible for writing and testing the CHITRA95 kernel, namely the core library, inserters and extractors. This permitted the rest of the CHITRA95 research group to focus on writing higher-layer modules, as the number of generators and presenters provided with CHITRA95 shows.

6.1.1 The cost of modularity

On the other hand, the modular design of CHITRA95 meant that certain compromises had to be made, particularly in the area of raw performance. The preceding generation, CHITRA93, kept all ensemble data in memory after translation from trace files. Table 6.1 shows timings for a number of operations by both CHITRA93 and CHITRA95 (timings for CHITRA93 are taken from [5]). The entire ensemble $\mathcal{E}$ consists of 10 separate traces taken from an execution of a multiprocessor implementation of a parallel database with record locking. All timings are in seconds. The dip in performance may be attributed to four main

Operation	CHITRA93	CHITRA95
Aggregate $\mathcal{E}$ by filter	17.0	640.8
Generate adjacency matrix of $\mathcal{E}$	3.0	490.4
Apply KW test to $\mathcal{E}$	9.0	20.4
Aggregate $\mathcal{E}$ by pattern — select and apply pattern	8.5	31.7
Undo transform	4.0	21.7
Generate 4 state semi-Markov model of $\{\mathcal{E}.0, \mathcal{E}.4, \mathcal{E}.7, \mathcal{E}.8\}$	5.0	663.8
Generate semi-Markov model of untransformed $\mathcal{E}$	28.5	1724.9

Table 6.1: Comparative timings (in seconds) for CHITRA93 and CHITRA95

factors:

- The cost of launching several distinct processes is typically higher than the cost of launching just one.
- The cost of pipelining, often implemented with disk buffering, is greater than the cost of accessing data already in the same address space or heap.
- The canonical format stream that facilitates data passing between modules requires that same data to be encoded and decoded as many times as there are modules.
- The cost of supporting the *undo* operation via the `Extents` mechanism which involves a logical-to-physical offset translation for each record access for ensemble versions > 0.

Of these four factors, the cost imposed by the `Extents` mechanism is most significant. Often, the extents lists that describe the members of the ensemble become lengthy due to insert and delete operations; this increased length raises the logical-to-physical offset translation time as a consequence. On the other hand, the cost can be alleviated by cloning the ensemble; the new ensemble is created at version 0 which dispenses with all offset translation.

6.2 New interface models interact with design decisions.

CHITRA95 was originally conceived primarily as a reimplementation of its immediate predecessor, CHITRA93, and as such was intended to provide a typical windowed graphical user interface with the standard set of menu and dialog box interactions. The advent of web trace analysis brought with it the desire to provide a World Wide Web browser-friendly interface; subsequently, the value of freeing the architecture from the windowed graphical user interface model to an interface model dependent solely on how the user interacts with the data was realized. In particular, work on CHITRA95 modules was initiated to implement their link to a user interface via the `CfgArgs` class, which implicitly assumes a windowed graphical user interface with menus and dialog boxes. Conversion work on the modules was interrupted when the need for a more abstract interaction model was realized.

6.3 Literate programming is wonderful if employed properly.

As mentioned elsewhere in this thesis, the discipline of literate programming was employed while writing the bulk of the source code and documentation for CHITRA95. Literate programming stresses the writing of clear, usable documentation as an integral part of the programming process as opposed to being an activity of secondary importance. If employed properly, it results in a readable, comprehensible document written with exposition and explanation in mind instead of a document whose structure follows the dictates of whatever programming language is used.

Our literate programming efforts produced mixed results that range from documents that read well from both documentation and coding points of view to documents that are little more than source code embedded in a skeletal FunnelWeb template without the benefit of accompanying documentation. These results came about because of our lack of experience in literate programming, the habit of coding as quickly as possible while deferring documentation to a later date and personal variations in expository skill. In the future, it would be advisable to begin by instituting policies and standards with respect to what may be considered a truly "literate" program; adjustments may subsequently be made as programmers/users gain experience.

6.4 Anticipate shifts in tool usage.

CHITRA was originally conceived as a tool for analyzing and visualizing time-series numerical traces; part of its repertoire was the notion of building a model of the trace with the aid of an editable visualization of the trace. Model building was made possible in great part by the ability to transform the trace such that the record space was greatly reduced without losing basic information about the behavior reflected by the trace. It was then assumed that all traces analyzed with CHITRA would be timestamped in some fashion; in order to facilitate the computation of statistics, absolute timestamp information was converted into and internally stored as occupancy times for each trace entry.

More recently, the use of CHITRA95 to analyze World Wide Web traces brought into question the usefulness of occupancy times and highlighted the need to maintain the original timestamps that came with the traces. On another front, increasing interest in the analysis of categorical data brought about the need to support trace records with non-numeric fields. These shifts in the nature of the traces analyzed with CHITRA95 instigated certain changes. For example, the extractor `e.all` which was originally intended to output trace entries in the form of (trace record encoding, occupancy time) pairs was augmented with the ability to output unencoded trace records and absolute timestamps instead. Indeed, the `Member`

class previously represented each trace entry as a (trace record encoding, occupancy time) pair; this was modified to permit absolute timestamps to be stored instead.

6.4.1 An embedded database manager can be useful.

Much of the information we have had to extract from the newer types of traces answer queries like how many times some feature occurs in a trace, what features occur in some specified subsequence of a trace, and what is the subsequence bounded by two timestamps. Clearly, these are queries that are commonly tackled by database management systems, suggesting the incorporation of one at the heart of CHITRA. By basing the ensemble representation upon a DBMS, we could gain leverage from the DBMS' performance and be able to specify queries in a high-level language (e.g., some modified form of SQL). On the other hand, the DBMS must be capable of retrieving sequences of records based upon a pattern (a fundamental requirement of the `x.a.pat.ens` transformer); furthermore, the cost of using CHITRA might be raised to an unacceptable level if a commercial DBMS were to be used.

6.4.2 Even subtle shifts in emphasis can have important repercussions.

The role of modeling and the use of transformers have both diminished with the shift in interest away from numerical time-series traces. Portions of the design reflect CHITRA95's modeling heritage, and much code is devoted to supporting version management, a direct consequence of the experimental approach to model building through the use of transformers. In particular, the support for the *undo* operation comes at a high cost, especially when a transformer has been applied to the ensemble. When coupled with the shift in emphasis away from modeling, it makes sense to consider revising the design to either split version management support away from core functionality or to drop the support entirely.

6.5 There is always room for design refinement.

Our experience with writing various CHITRA95 components suggests several refinements that might be made to the design. First, instead of having two distinct interfaces to the ensemble representation, namely the `Ensemble` class interface and the `CFStream` class interface, it is perhaps desirable to have only one. Having a unified access model would flatten the learning curve associated with writing inserters and extractors, and basing the model on `CFStream` would stress the notion of streaming data from one CHITRA95 module to another. Ideally, any such unified interface would preserve both the random-access capability afforded by the `Ensemble` class and the more elegant operator notation used by the `CFStream` class.

Second, the `Ensemble` class currently provides an interface to the `Member` and `StateMap` instances that it maintains by providing methods that essentially duplicate the ones already provided by the `Member` and `StateMap` classes. A simpler approach would be to return references to these instances instead where appropriate; by doing so, we can stress the distinction between member-level and ensemble-level operations.

Third, the data representation used by the `Member` class stores trace record encodings instead of trace records. Currently, all trace records are mapped to trace record encodings at the `Ensemble` level; the encodings are then passed on to the `Member` method call. This imposes a potential parallelization bottleneck. Instead, a reference to the `StateMap` instance should be passed to the `Member` which then performs the necessary trace record mapping calls; the `StateMap` instance would then be responsible for arbitrating mapping calls from different `Member` instances.

Chapter 7
CONCLUSIONS

We conclude by summarizing the results of this thesis. We then suggest possible directions for future development.

7.1 Thesis Summary

The design of CHITRA95 had five specific goals: extensibility, reusability, portability, reliability and maintainability. CHITRA95's layered design and reliance upon a text-based streamable, canonical data format permit new modules to be developed concurrently in the user's programming language of choice. Extensibility in another sense, that of being able to contribute functionality to another tool, is again made possible by CHITRA95's relatively small module granularity; portions may be extracted and used at will as opposed to having to integrate an entire tool with another. Portability is enhanced not just by attention to using portable building blocks, but also by separating computation from display, thus freeing most module writers from having to write to a specific implementation of a specific human-computer interaction model. Reliability is promoted by the ability to write simple, incremental tests that compare expected against actual module output (i.e., regression testing). Finally, the use of small modules written in a familiar programming or scripting language, in tandem with the literate programming methodology, clearly promotes maintainability. CHITRA95's success at achieving these goals is borne out by the collective experience of the CHITRA research group.

7.2 Future Directions

Interest in CHITRA95 has been expressed by researchers from a major on-line service, as well as investigators from the Technical University of Delft in the Netherlands and the University of California at Los Angeles. The recent explosion in interest in the World Wide Web and in related technologies suggests several possible areas of further study, as follows:

- *Navigating traces in a virtual reality.* As we begin to explore larger traces and even larger ensembles, it becomes increasingly difficult to identify features of interest because of the bandwidth limitations of a two-dimensional display such as a graph on a piece of paper. Exploring traces in three dimensions could prove useful, particularly in an immersive environment. In particular, we suggest adding a viewer that is part of a virtual reality environment, such as a virtual reality CAVE [7, 19]. Research into supporting non-graphical user interface input, such as voice or gestures, is also indicated.

- *The impact of the Java programming language.* Using the Java programming language [9] to implement CHITRA would provide a new level of portability because Java itself is designed to be machine-independent, operating system-independent and graphical user interface toolkit-independent. It would permit CHITRA to be easily downloaded over the Internet without the need to compile source code. Moreover, Java comes with a useful graphical interface library that is also portable to the two dominant windowing systems today, namely X/11 and Microsoft Windows. On the other hand, Java is a new language and its application programming interfaces are still very much in flux. As with any new product, a number of bugs and limitations still need to be worked out before serious development in it can proceed.

- *Collaborative navigation.* At the moment, CHITRA is strictly a single-user trace analysis tool. The benefit of having two or more investigators in multiple geographic locations analyze the same trace at the same time is obvious: features of interest not

readily apparent to one might be pointed out by another; the need for travel would be obviated; discussion of a trace in a group could be made more fruitful and entertaining by providing each member with the means to actively participate as opposed to simply following the discussion leader [13].

- *Data mining.* Trends in large amounts of data are often difficult to discern. A future version of CHITRA might provide automated means of searching for and identifying these trends instead of relying solely on the user's ability to discover these trends on her own.

- *Joint cognitive navigation.* As the computers that host CHITRA become more powerful, it makes increasing sense to use that power to aid the user in navigating large traces. It is almost certain that the human brain will remain the superior tool for discovering patterns and extracting knowledge from trace data. By combining this with a computer's ability to filter out extraneous data and to suggest possible areas of interest discovered through programmed heuristics, we can achieve a degree of synergy between the CHITRA user and the CHITRA system itself.

REFERENCES

[1] M. Abrams. CHITRA93*: A System to Model Ensembles of Trace Data — User Manual.* Computer Sci. Dept., Virginia Tech, Blacksburg, VA 24061-0106, Aug. 1994. <URL: http://www.cs.vt.edu/~chitra>.

[2] M. Abrams. CHITRA95 — *User Manual.* Computer Sci. Dept., Virginia Tech, Blacksburg, VA 24061-0106, May 1996. <URL: http://www.cs.vt.edu/~chitra>.

[3] M. Abrams, A. Batongbacal, R. Ribler, and D. Vazirani. CHITRA94: A tool to dynamically characterize ensembles of traces for input data modeling and output analysis. Technical Report TR 94-21, Computer Sci. Dept., Virginia Tech, Blacksburg, VA 24061-0106, June 1994. <URL: http://www.cs.vt.edu/~chitra>.

[4] M. Abrams, N. Doraswamy, and A. Mathur. Chitra: Visual analysis of parallel and distributed programs in the time, event, and frequency domain. *IEEE Trans. on Parallel and Distributed Systems*, 3(6):672–685, Nov. 1992.

[5] M. Abrams, T. Lee, H. Cadiz, and K. Ganugapati. Beyond software performance visualization. *Concurrency – Practice and Experience*, 7(8):737–764, Dec 1995.

[6] M. Abrams, S. Williams, G. Abdulla, S. Patel, R. Ribler, and E. A. Fox. Multimedia traffic analysis using Chitra95. In *Proc. ACM Multimedia '95*, pages 267–276, San Francisco, Nov. 1995. ACM.

[7] C. Cruz-Neira et al. Scientists in wonderland: A report on scientific visualization applications in the cave virtual reality environment. In *Proceedings of the IEEE Symposium in Research Frontiers in Virtual Reality*, San Jose, CA, Oct. 1993. Visualization 93.

[8] N. Doraswamy. Chitra: A visualization system to analyze the dynamics of parallel programs. Master's thesis, Computer Sci. Dept., Virginia Tech, Blacksburg, VA 24061-0106, Dec. 1991.

[9] D. Flanagan. *Java in a Nutshell: A Desktop Quick Reference for Java Programmers.* O'Reilly & Associates, Sebastopol, CA, Feb. 1996.

[10] S. Gundavaram. *CGI Programming on the World Wide Web.* O'Reilly & Associates, Sebastopol, CA, Mar. 1996.

[11] D. E. Knuth. Literate programming. In D. E. Knuth, editor, *Literate Programming*, pages 99–136. Center for the Study of Language and Information, Stanford Univ., 1992.

[12] D. Lea. *User's Guide to the GNU C++ Library.* Free Software Foundation, Inc., Apr. 1992.

[13] J. Leigh, A. Johnson, C. Vasilakis, and T. Defanti. Multi-perspective collaborative design in persistent networked virtual environments. In *Proceedings of the VRAIS 96.* IEEE, 1996. Still to appear.

[14] A. D. Malony, D. H. Hammerslag, and D. J. Jablonowski. Traceview: A trace visualization tool. *IEEE Software*, 8(5):19–28, Sept. 1991.

[15] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison Wesley, Reading, MA, 1996.

[16] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, Reading, MA, 1994.

[17] D. A. Reed, R. A. Aydt, T. M. Madhyastha, R. J. Noe, K. A. Shields, and B. W. Schwartz. The Pablo performance analysis environment. Dept. of Comp. Sci., Univ. of IL, 1992.

[18] R. Ribler, A. Mathur, and M. Abrams. Visualizing and modeling categorical time series data. In *Symposium on Visualizing Time-varying Data*, volume NASA Conference Publication 3321, Williamsburg, VA, Jan. 1996. ICASE and NASA/LaRC. <URL: http://www.cs.vt.edu/~chitra/docs/95vtvdRMA/95vtvdRMA.html>.

[19] T. M. Roy, C. Cruz-Neira, and T. A. DeFanti. Cosmic worm in the cave: Steering a high-performance computing application from a virtual environment. *To be published in the special issue on Networks and Virtual Environments of Presence: Teleoperators and Virtual Environments*, Fall 1994.

[20] P. H. Salus. *A Quarter Century of UNIX.* Addison Wesley, Reading, MA, 1994.

[21] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison Wesley, Reading, MA, 2nd edition, 1991.

Appendix A

THE Chitra95 KERNEL API

The application programmer interface to the CHITRA95 kernel is dominated by the interface to two C++ classes: `Ensemble` and `CFStream`. Any new module that needs to work with the persistent ensemble representation must work in terms of class `Ensemble`; similarly, if the new module is to communicate with other CHITRA95 modules as seamlessly as possible, it needs to do so via the services provided by class `CFStream`.

This section is organized as a graduated tutorial in which more detail is presented with such successive lesson. Each tutorial section begins by giving the prototype for each class method used in the section.

A.1 Opening and closing an ensemble

```
Bool Ensemble::Open(String fname, String mode = "r");
Bool Ensemble::Close(void);
```

An `Ensemble` instance is declared just like any other variable. Once declared, it may be opened by calling its `Open` method, as in the following code fragment:

```
Ensemble e;
if (!e.Open("mydata.ens", "r")) {
    // complain about the failure and exit
}
```

A mode of `"r"` means that the ensemble is to be opened in read-only mode; this mode is used by default if no mode is specified. The only other valid mode is `"rw"` which specifies that the ensemble is to be opened in read-write mode.

An open ensemble can be closed at any time by calling the `Close` method:

```
e.Close();
```

A.2 Adding a new member

```
Bool Ensemble::AddMember(
                    String sourceTraceName = ‘‘’’,
                    Member::member_t mbrType = Member::MOT,
                    Member::timestamp_t tsType = Member::OCCUPANCY_TIME,
                    Time startTime = 0,
                    Time endTime = 0,
                    Time timeIncrement = 0);
UInteger Ensemble::NumMembers(void);
```

Before member data can be examined or modified, that member must already exist. Each member is uniquely identified by an integer from 0 to $N-1$ where N is the number of members currently in the ensemble. The value of N can be obtained by using the `NumMembers` method:

```
UInteger N;
N = e.NumMembers();
```

If a new member needs to be added to the ensemble for any reason (e.g. N is zero), the `AddMember` method must be invoked. This method creates a new member with the specified characteristics; if successful, the member is immediately accessible with id $N-1$. The *mbrType* and *tsType* arguments may be given either their default values as listed in the prototype above or the values `Member::M` and `Member::ABSOLUTE_TIME`, respectively. A value greater than zero for the *timeIncrement* argument means each record in the member has the same occupancy time, namely *timeIncrement*. The following example adds a new member with the associated source trace name `"newtrace.cf"`.

```
if (N == 0) {
    e.AddMember("newtrace.cf");
    ++N;
}
```

A.3 Reading member data

```
MState Ensemble::ModelState(Cursor pos, UInteger mbrNum);
```

```
Time Ensemble::OccupancyTime(Cursor pos, UInteger mbrNum);
Time Ensemble::AbsoluteTime(Cursor pos, UInteger mbrNum);
Member::timestamp_t Ensemble::TimestampType(UInteger mbrNum);
UInteger Ensemble::Length(UInteger mbrNum);
```

As described elsewhere, each member record is composed of a trace record encoding, otherwise known as a model state and an optional timestamp. CHITRA95 time stamps are either occupancy times or absolute time stamps; the timestamp type for each member can be determined by calling the `TimestampType` method:

```
if (e.TimestampType(N-1) == Member::OCCUPANCY_TIME) {
    // do stuff with occupancy times
} else if (e.TimestampType(N-1) == Member::ABSOLUTE_TIME) {
    // do stuff with absolute times
}
```

The `ModelState` and `AbsoluteTime` methods can be called for member i for any value of pos from 0 to $L-1$ where L is the length of member i (i.e. the value returned by `Ensemble::Length(i)`). If the timestamp type of member i is `Member::OCCUPANCY_TIME`, the `OccupancyTime` method can also be validly called for the same values of pos. If the timestamp type of member i is `Member::ABSOLUTE_TIME`, the `OccupancyTime` method can be validly called for the same member only for positions 0 through $L-2$, inclusive. This is because the occupancy time of item p in this instance is actually the difference in absolute time stamps between items $p+1$ and p.

A.4 Appending new records to a member

```
Bool Ensemble::Append(MState ms, Time ts = 0, UInteger mbrNum);
```

Regardless of what the current ensemble version may be, the data tuple *(ms,ts)* can always be appended to member i, provided that member i exists, by invoking the `Append` method as follows:

```
e.Append(ms, ts, i);
```

A.5 Starting a new version

```
UInteger Ensemble::Version(void);
Bool Ensemble::StartNewVersion(void);
```

The `Version` method returns the current version of the ensemble; this will be some number $v \geq 0$. Before any member of the ensemble can be modified with the `Insert`, `Delete` or `Collapse` methods, a new version must be started (i.e. the ensemble version must be greater than zero) so that the modifications can be rescinded if needed. In general, a new version must be started immediately preceding a block of ensemble edit operations that may need to be undone as a block in the future.

```
if (e.Version() == 0) {
    if (e.StartNewVersion()) {
        // do edits here
    }
}
```

A.6 Editing member data

```
Bool Ensemble::Insert(Cursor pos,
                      MState ms,
                      Time ts = 0,
                      UInteger mbrNum);
Bool Ensemble::Delete(Cursor pos,
                      UInteger n = 1,
                      UInteger mbrNum);
Bool Ensemble::Collapse(Cursor pos,
                        UInteger n,
                        MState ms,
                        UInteger mbrNum);
```

If L is the length of ensemble member i, then the (model state, timestamp) pair *(ms,ts)* can be successfully inserted at position p where $0 \leq p < L$. Also, up to n tuples may be deleted from ensemble member i starting at position p, provided that $0 \leq p \leq L - p$. Finally, a contiguous subsequence of n tuples starting at position p may be collapsed

into a new tuple *(ms,T)* provided that $0 \leq p \leq L - p$. Here, the timestamp T is either the cumulative occupancy time of the original subsequence if the member's timestamp type is `Member::OCCUPANCY_TIME`; otherwise (i.e. the member's timestamp type is `Member::ABSOLUTE_TIME`), T is the timestamp of the first tuple in the original subsequence.

A.7 Undoing changes to the ensemble

```
Bool Ensemble::Undo(UInteger numVersions = 1);
```

As described previously, the `Version` method returns some number v which is called the current version of the ensemble. The `Undo` method can then be used to undo up to v versions, although the typical usage of Undo is to undo only the most recent version (i.e. the current version).

```
if (e.Version() > 0) e.Undo();
```

A.8 Computing ensemble and ensemble member statistics

```
Set<UInteger> Ensemble::Contains(MState ms);
Bool          Ensemble::Contains(MState ms, UInteger mbrNum);
UInteger      Ensemble::Count(MState ms);
UInteger      Ensemble::Count(MState ms, UInteger mbrNum);
Time          Ensemble::Duration(MState ms);
Time          Ensemble::Duration(MState ms, UInteger mbrNum);
MState        Ensemble::MaxModelState(void);
MState        Ensemble::MaxModelState(UInteger mbrNum);
UInteger      Ensemble::NumUniqueModelStates(UInteger mbrNum);
UInteger      Ensemble::Length(void);
```

The `Contains` method determines whether a given model state appears in any member of the ensemble or alternately, in a specific member of the ensemble, depending on whether a member id is passed in. Analogously, two variants each of the `Count`, `Duration` and `MaxModelState` methods are available; respectively, they determine how many times a given model state appears, how much total occupancy time it accounts for and which is the highest numbered model state. The `NumUniqueModelStates` method returns the number of unique

model states in the specified member. The `Length` method, if passed no arguments, returns the total length of the ensemble (i.e. the sum of the length of each of the ensemble's members).

A.9 Other ensemble operations

```
Time Ensemble::StartTime(UInteger mbrNum);
Time Ensemble::EndTime(UInteger mbrNum);
Time Ensemble::TimeIncrement(UInteger mbrNum);
```

The `StartTime` method returns the absolute timestamp of the first data tuple of the specified member. In a similar fashion, the `EndTime` method returns the entry time of the last data tuple of the specified member if and only if the member contains absolute time stamps; on the other hand, if the member contains occupancy times, the value returned is the entry time of the last data tuple plus the occupancy time of that same tuple. The `TimeIncrement` method returns a non-zero value if and only if the data tuples of the specified member contain identical occupancy times.

A.10 Input/output: `CFStream` programming concepts

The `CFStream` class provides an interface to a data stream in CHITRA95 canonical format. Such a data stream contains a sequence of N objects numbered 0 through $N-1$. Each object may be of a distinct type. The two object types encountered most frequently in CHITRA95 are *sequence* and *matrix*. `CFStreams` also often three object types that carry definitions of trace records and trace record encodings: *system state component dictionary*, *system state component value dictionary* and *system to model state mapping*[1].

Objects are separated by a logical *end-of-object* marker, and the stream itself is terminated by a logical *end-of-stream* marker. A `CFStream` instance typically reads from standard input and writes to standard output; however, different input and output streams may be specified in the `CFStream` constructor.

[1] A *system state* is what CHITRA93 calls a trace record, excluding any timestamp.

Each object in a `CFStream` may be associated with one or more *attributes*, an attribute being a simple (name,value) pair. The data stream may also contain attributes not associated with any object in particular; such attributes are *global*.

All data in an object is organized into *tuples* of uniform length and composition. Each tuple element is of one of the CHITRA95 scalar types; structured elements are not supported. The `CFStream` user is responsible for determining the tuple composition, either through prearrangement or by interpreting the attributes and data contained in a system state component dictionary object, if one is provided.

The first read operation on a `CFStream` reads from object 0, as long as data is present and the end of the object has not been reached. Once end of the current object is reached, all further reads will fail until the input is explicitly advanced to the next object by means of a `CFStream` method call and doing so does not cause the end of the stream to be reached. The input may also be advanced to the end of the current object at any time.

One important fact to remember about canonical format streams is that all attributes appear in the stream prior to any data tuple. This means that any attribute for any object may be read at any time from an input stream, even if the current input pointer has not reached the object yet. This also means that all `CFStream` method calls to write attributes to a canonical format stream must precede any method call to write a data tuple element.

The next few sections illustrate the usage of the most common `CFStream` methods. More comprehensive examples may be found in the source documents *a.t.cf2ens.fw* and *e.all.fw*.

A.11 Reading and writing attributes

```
Bool CFStream::GetAttribute(const char* attrib,
                            char*&      returnValue,
                            Integer     objectNum = -1);
Bool CFStream::GetAttribute(const char* attrib,
                            UInteger&   returnValue,
                            Integer     objectNum = -1);
Bool CFStream::GetAttribute(const char* attrib,
                            Real&       returnValue,
```

```
                                Integer     objectNum = -1);
    Bool CFStream::CheckAttribute(const char* attrib,
                                  const char* value,
                                  Integer     objectNum = -1);
    Bool CFStream::CheckAttribute(const char* attrib,
                                  UInteger    value,
                                  Integer     objectNum = -1);
    Bool CFStream::CheckAttribute(const char* attrib,
                                  Real        value,
                                  Integer     objectNum = -1);
    Bool CFStream::PutAttribute(const char* attrib,
                                const char* value,
                                Integer     objectNum = -1);
    Bool CFStream::PutAttribute(const char* attrib,
                                UInteger    value,
                                Integer     objectNum = -1);
    Bool CFStream::PutAttribute(const char* attrib,
                                Real        value,
                                Integer     objectNum = -1);
```

The `GetAttribute` method copies the value of the attribute named *attrib* associated with the object numbered *objectNum* into *returnValue*, if such an attribute is defined. It returns TRUE if and only if successful. If *objectNum* is -1 or omitted, *attrib* is interpreted as a global attribute.

The `CheckAttribute` method is analogous to `GetAttribute`; it returns TRUE if and only if the specified attribute has the specified value.

The `PutAttribute` method associates an attribute (name,value) pair with a specific object if *objectNum* $\geq$ 0 or with the stream as a global attribute if *objectNum* is -1 or omitted, provided that no data tuple element has been output to the stream. It returns TRUE if and only if this provision is satisfied and no other errors occur.

```
CFStream cfs;
UInteger numObjects;
char     objectType[80];

// Get number of objects in input stream
if (cfs.GetAttribute(AN_NUMOBJS, numObjects)) {
```

```
        cerr << "Input stream contains " << numObjects
             << " object(s)" << endl;

        for (UInteger i = 0; i < numObjects; ++i) {
            // Get type of each object in input stream
            if (cfs.GetAttribute(AN_OBJTYPE, objectType, i)) {
                cerr << "Object " << i << " is of type "
                     << objectType << endl;
            }

            // Make sure object is of type "sequence"
            if (!cfs.CheckAttribute(AN_OBJTYPE, AV_OBJTYPE_SEQUENCE, i)) {
                cerr << "Object " << i << " is not a sequence!" << endl;
            }
        }
    }

    // Start sending attributes to the output stream
    cfs.PutAttribute(AN_NUMOBJS, 1);
    cfs.PutAttribute(AN_OBJTYPE, AV_OBJTYPE_SEQUENCE, 0);

    // Send data tuples at this point
    ...
```

For a complete list of predefined attribute names and values, refer to the `CFStream` source document *cfstream.fw*.

A.12 Reading and writing data tuples

```
CFStream& CFStream::operator >>(type& val);
CFStream& CFStream::operator <<(type val);
CFStream& CFStream::endr(CFStream& cfs);
Bool      CFStream::PutEndOfObjectMarker(void);
void      CFStream::AdvanceToNextObject(void);
void      CFStream::SkipToEndOfObject(void);
Bool      CFStream::EndOfObject(void);
Bool      CFStream::EndOfStream(void);
int       CFStream::operator !();
```

Data tuples are read from and written to a `CFStream` one element at a time using the

`>>` and `<<` operators. When writing a data tuple, the end of the tuple is marked with the `endr` manipulator. Once all data tuples for an object have been written, the end of the object is marked by calling `PutEndOfObjectMarker`.

```
MState enc;  // trace record encoding
Time   ts;   // timestamp

cfs >> enc >> ts;
++enc;
ts = 0;
cfs << enc << ts << endr;

// Output rest of object, then mark end
cfs << ... ;
... ;
cfs.PutEndOfObjectMarker();
```

When reading in the data tuples from the objects in the stream, use `EndOfObject` and `EndOfStream` to control a suitable loop. Remember that `AdvanceToNextObject` must be called when the end of the current object is reached. In case of error, the rest of the object can be skipped by calling `SkipToEndOfObject`.

```
while (!cfs.EndOfStream()) {
    while (!cfs.EndOfObject() {
        // read in data tuples and do stuff
        ...

        // check for error
        if (!cfs) cfs.SkipToEndOfObject();
    }
    cfs.AdvanceToNextObject();
}
```

VITA

Alan L. Batongbacal was born in Quezon City, Republic of the Philippines on May 4, 1965. In addition to his hard-won master's degree in Computer Science, he holds baccalaureate degrees in Physics and Computer Engineering from the Ateneo de Manila University. Prior to leaving the Philippines for his studies at Virginia Tech, he held various consulting positions and taught computer science subjects at Ateneo. Currently employed as a software engineer for Recognition Research, Inc. in Blacksburg, Virginia, he maintains a professional interest in operating systems, networks and software development. He spends his free hours watching anime ("Banzai Ikkoku-kan!") and indulging a growing passion for things Japanese. He may be reached via e-mail at `alanlb@rrinc.com` or at `alanlb@cs.vt.edu`.