The Design and Implementation of CHITRA92, a System to Empirically Model Concurrent Software Performance

by

Krishna Ganugapati

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:

[illegible]

Dr. Marc Abrams, Chairman

[illegible]

Dr. Dennis G. Kafura

[illegible]

Dr. Calvin J. Ribbens

April, 1993

Blacksburg, Virginia

LD
5655
V855
1993
G368
Spec.

Spec

The Design and Implementation of CHITRA92, a System to Empirically Model Concurrent Software Performance

by

Krishna Ganugapati

Committee Chairman: Dr. Marc Abrams

Computer Science

(ABSTRACT)

With parallel and distributed computing entering the mainstream of computer science, it is important to ensure that parallel application codes are optimized to achieve the best possible performance. This thesis describes the design and implementation of CHITRA92, the second generation of a performance analysis system for parallel programs. CHITRA92 is unique in that it uses visualization techniques to analyze the dynamic activity of a program and produces a semi-Markov chain model of the program's behavior. This model can be parameterized to predict behavior of a program and identify the performance bottlenecks in the program. The important contributions of the CHITRA92 system are:

- The dynamic activity of a program is represented through a *program execution sequence* (PES) and a set of *program parameters.* A PES description language has been defined to allow users to describe the structure of a PES *state vector* and instances of the state vector.

- A PES is reduced to a semi-Markov chain model via a set of transformations.

- Visualization is used to assist in selecting which transforms to apply to a PES.

- The presence of periodic behavior in PES's can be identified using spectral analysis techniques.

ACKNOWLEDGEMENTS

I am deeply indebted to Dr. Marc Abrams, my advisor, for his constant guidance and support throughout my graduate career. He has helped me in countless ways. He is more than an advisor to his students, he is a friend to all of us.

I am grateful to Drs. Dennis Kafura and Cal Ribbens for their patience and flexibility when it came to reading this thesis and serving on my thesis committee. They have made many useful suggestions both in the final defense as well as in the manuscript.

I cannot begin to express my gratitude to members of the Chitra group for their camaraderie and friendship. Special thanks are due to Tim Lee who helped with portions of the implementation. Alan Batongbacal cheerfully read through my final draft and I'm grateful to him for this.

I've had the good fortune of making many great friends at Virginia Tech. I'd like to thank all of them for making my stay here a memorable one.

I am grateful to the Department of Computer Science, Virginia Tech, and to the National Science Foundation (grant CCR-9211342) for financial assistance in the form of teaching and research assistantships.

And finally, I thank my parents, my sister Rama and my brother-in-law Sriram, but for their encouragement, love, and support, I could never have become what I am today.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
Introduction

Massively parallel computer systems are rapidly replacing traditional vector supercomputers as viable means to solving large numeric and scientific problems. The next generation of parallel supercomputers will be predominantly high performance MIMD architectures having potential performance that exceed three hundred gigaflops [22]. Connectionist architectures are predicted to cross the teraflop barrier within the next decade [2].

With the advent of massively parallel distributed systems, it has become more important than ever to optimize parallel applications so that they exploit maximum possible parallelism while making a conservative use of system resources to execute at peak speeds.

Performance analysis techniques require

- defining *performance metrics*: these are criteria by which the performance of a program is evaluated. Examples of performance metrics for a parallel program are its speedup, its processor utilization, and its cache or primary memory reference patterns.
- locating *performance bottlenecks*: these are the causes for unsatisfactory performance in a program.
- redesigning an application program to eliminate the performance bottlenecks or at the very least, minimize their effect.

Typically, the program designer is interested in optimizing one or more performance metrics, and seeks a configuration of the application that best satisfies these metrics. The process of identifying a configuration that optimizes a set of performance metrics for an application is called *fine-tuning*. While badly designed applications can waste substantial amounts of computing resources, even correct programs need to be fine-tuned for maximum efficiency. It is also important to analyze the performance of an application across multiple computer systems to determine a suitable hardware configuration for the application. From an application's standpoint, the objective of performance analysis is to understand how exactly the application's design and the underlying system's hardware influence the application's performance.

At present, no generally accepted method exists that can analyze the performance of an application on parallel architectures. Some of the major stumbling blocks in characterizing the behavior of parallel application are the following:

- The difference between the performance analysis of parallel applications and sequential applications is that there exists the need to observe dynamic parallel behavior. In the case of sequential applications, execution profilers can often correctly identify bottlenecks in the program's source code. This is not necessarily true in the case of parallel applications, where the interaction of multiple threads of execution is the primary reason for bad program performance.

- Advances in parallel system design have significantly increased the number of performance-related parameters, such as the number of concurrently executing processes, or the resources that are acquired by an executing application. Also, due to the diversity in architectures and programming paradigms, these parameters rarely correspond between architectures.

- A program is said to show *nondeterministic* behavior if different runs of the program may execute statements in different orders, consume different amount of resources and even produce different outputs. The sequence of statements executed in the single run of a program is called its *program execution trajectory.* Parallel programs are inherently nondeterministic because concurrently executing processes within the program usually compete for resources. This possibility of having different execution trajectories for the same application is another reason why program profiling is inadequate for analyzing parallel applications.

1.1 Extant Performance Analysis Techniques

As mentioned earlier, no general method of performance analysis for parallel and distributed applications is available. There are a variety of techniques that claim some measure of success in solving the analysis problem. We can broadly classify them as follows: theoretical techniques, experimental techniques, and hybrid analysis. Sections 1.1.1, 1.1.2, and 1.1.3 discuss each of these techniques in detail.

1.1.1 Theoretical Techniques

Most theoretical techniques build a model of the application's behavior, given the program's source code and a set of data items, referred to as the program's *parameter set.* Examples of parameters are the number of executing processes in the program or a resource allocation policy. The model is solved by numerical or simulation techniques and then subjected to analysis. Major modeling techniques used include queueing networks [9], stochastic processes [43], timed or stochastic Petri nets [10, 19], and simulation models [9]. Performance metrics are computed from the model and then experimentally verified. Often different models are constructed for different input parameter sets and then integrated together to form a parameterized version of the model. For a given input parameter set, the model's accuracy is determined by comparing its predictions of a metric's value to the actual computed value of the metric.

In [9], Abrams discusses why results from theoretical models do not justify the effort that goes into constructing and then solving the model. First, it is usually cumbersome to model parallel applications (an exception to this is found in [6]). It is next to impossible to attempt any kind of scaling on the size of input supplied to the program or on the number of available processing units because the time taken to solve the model is usually exponential with respect to these parameters. Second, the model needs to be solved for every distinct set of inputs and parameter values. Third, only limited classes of software can be modeled.

1.1.2 Empirical Techniques

The second approach to performance analysis of parallel applications that is fast gaining acceptance in the research community is more intuitive than analytical. First, the dynamic parallel activity of an application is captured by instrumenting the program's source code. The resulting program execution trace is then visualized. More often relationships between data that are fairly obscure become apparent when perceiving them graphically. The advantages of using graphics to display performance of parallel programs can be seen in JED [26], Hyperview [28], IPS-2 [38] and several other performance visualization environments. The insights obtained into program behavior are comparable to those obtained from traditional modeling techniques.

This is not to say that visualization of dynamic parallel activity does not have its disadvantages. The diversity of programming systems and architectures has resulted in no standard implementation or analysis methodology for visualization systems. This has encouraged the isolated development of tools that are too narrowly focused to have any kind of general applicability. Secondly, the volume of trace data generated makes it very hard to store and manipulate for analysis. This problem is further compounded by the fact that there is the possibility of different program trajectories. The interpretation of the application's performance from the study of a single execution trace does not necessarily hold for other traces of the same program's execution. Thirdly, when time dependent data is used, there is rarely any linking of the data to the source code that generated the data.

The single, biggest hurdle that empirical techniques face is the lack of a systematic methodology to compute a performance metric from a trace. Another drawback of empirical techniques is their tendency to focus on a single aspect of a program's behavior and as such cannot present a global picture of the program's behavior.

1.1.3 Hybrid Analysis

Both empirical and theoretical analysis techniques have desirable properties that complement one another. Theoretical techniques are tools that generate models of program behavior. Their primary disadvantage lies in their inability to represent nondeterministic behavior associated with programs. Experimental techniques such as visualization lack formal methodologies to tune programs or predict their performance. They allow program data to be represented using graphical images and rely on the perception of the analyst to draw conclusions.

Visualization can be used effectively as a tool to assist in the theoretical modeling process. Transforms can be defined based on visual presentations of a trace. Where the execution trace of a program is difficult to comprehend and evades modeling, its structure can be transformed into a simpler, more regular approximation that permits modeling. Sections of the program trace that are deterministic in their behavior can be modeled separately and then assembled together to form the complete model. Another example of a powerful visual analysis technique is to compute and display a periodogram of the discrete Fourier transform of an execution trace to identify periodicity in the trace.

This thesis is founded on the premise that it is appropriate to use a hybrid of empirical and theoretical techniques in a methodology to analyze the behavior of a parallel program. Rather than simply visualize an execution trace of a parallel program and hope to intuitively understand the behavior of a metric, we construct a model of the parallel program from multiple observations of the execution trace of the program for a given input parameter set(s). The constructed empirical model should then be able to generate an execution trajectory that closely matches an observed program trajectory. Another measure of accuracy of the model is that metrics derived from the program model should closely match metrics derived from an actual observation of the program.

There are several advantages in using a hybrid of theoretical and empirical performance analysis techniques.

- The cost incurred in building a model of a program behavior for analysis is minimized. This is because the observed execution trace that is input to the empirical model construction process is itself a model of the program's behavior.

- Visualization assists in partitioning the execution trace and studying sections of the trace separately. Model construction can be done on separate components and then integrated.

- Visual comparison of multiple execution traces (for the same or different input parameter sets) is useful in both verifying similarities in traces and generating a parameterized model.

1.2 Problem Statement

In his thesis [16], Doraswamy explored the possibility of using visualization to analyze parallel program behavior. As part of this effort, an experimental prototype system *Chitra91** was developed.

This thesis discusses the design and implementation of CHITRA92, a second generation performance analysis system. The system is given a program and a set of execution traces, each the result of monitoring the program's execution for some parameter set. The system generates an empirical model of the program's behavior. The system also investigates whether any form of deterministic behavior exists in the program being monitored. The Chitra system therefore integrates performance analysis with visualization techniques to generate empirical models of program behavior.

The design and implementation of the CHITRA92 system is the first step towards the long term goal of the construction of a system which, given program traces collected from multiple program execution observations, traces back from a graph of a performance metric to sections of source code responsible for causing that behavior (as specified by the metric).

*Chitra is the Sanskrit word for pleasing paintings or pictures.

The efficacy of the system rests on how accurately the source code causing a bottleneck could be identified starting from a metric identifying the bottleneck.

1.3 Contributions

The contributions of this thesis are as follows.

Portable Trace Format: One of the biggest problems in the development of performance analysis systems is the unavailability of portable performance analysis tools across different platforms, largely due to the lack of portable performance data traces. The CHITRA92 system provides a trace description language that allows trace data to be represented in text and binary formats.

Visualization as a means to modeling: Most performance analysis systems consider visualization to be synonymous with performance analysis. Chitra is unique in that it uses visualization as a means to reduce a trace to a semi-Markov chain model. Six different displays have been constructed to visualize an execution trace: X-Y plots in the time and event domains, text displays, periodogram displays and Markov chain displays.

Transformations: The CHITRA92 system provides a set of powerful transforms that take an execution trace as input and generate a modified version of the execution trace which is more amenable to modeling. There are four different kinds of transforms: filter transforms to identify and remove random behavior in a trace, pattern aggregation transforms that identify and aggregate patterns, clipping transforms to identify the steady-state in a trace, and user-defined projection transforms.

Identifying Periodicity: The CHITRA92 system uses spectral analysis techniques to identify periodic behavior in performance data trace by constructing periodograms in the time and event domains.

Model Generation: The CHITRA92 system generates a discrete-space, homogeneous, semi-Markov stochastic process model [12, 42] of an execution trace. This semi-Markov chain model can be parameterized to predict program behavior at different input parameter sets and, consequently, identify performance bottlenecks.

The rest of this thesis is organized as follows. Chapter 2 surveys other performance analysis systems and briefly discusses the role of visualization in parallel program performance analysis. Chapter 3 introduces program execution sequences (PES's), and semi-Markov chains. A discussion of the methodology employed to transform an execution trace to a model or an ensemble of execution traces into set of models is also presented. Chapter 5 discusses the design goals of the system and provides a functional description. Chapter 6 contains conclusions and discusses the future direction of the Chitra project.

Chapter 2
Related Work

There has been a substantial amount of work on performance analysis of parallel systems and software in the last decade [25]. Visualization has been identified as one of the more prominent tools to be used for performance analysis, so much so that there exists the notion that performance visualization is synonymous with performance analysis studies. The Chitra perspective is slightly different. We believe that visualization is extremely useful in condensing performance data to make it amenable to analysis techniques. Section 2.1 gives a broad overview of the state of the art in performance visualization. The Chitra perspective on visualization is then presented.

The CHITRA92 performance analysis environment is unique with respect to its fundamental goal of modeling performance data and prescribing a methodology to generate a model of a program's behavior. There have been a number of performance analysis systems developed in the last decade that have influenced the development of the Chitra system. Sections 2.2 to 2.7 review four systems that have heavily contributed to the design and implementation of CHITRA92: IPS-2 [38], MemSpy [29], JED [26], PIE [30], Mtool [31], and Pablo [22]. Each section describes the objectives of the system and gives a brief description. Finally, we evaluate these systems with respect to CHITRA92 on different criteria such as the sophistication of the interface, the functionality of the system, the level at which information is presented, and the tools supported by the system.

2.1 The Role of Visualization in Performance Analysis Systems

Performance studies have always been tied to the design and implementation of large hardware and software systems. Not only is performance analysis essential for benchmarking parallel systems, it is invaluable in providing feedback that influences the development of new generations of these systems. A serious problem that handicaps the performance analysis process for parallel systems and software is the volume and complexity of performance data generated by these system. Sorting through and understanding this performance data is a daunting task. There are two basic problems: first, there must be a way by which an analyst can inspect the data itself, and secondly the data itself needs to be analyzed. The first problem arises because of the volume of performance data that can be generated by a parallel system or application. The second problem, analysis, requires modeling the experimental data so as to understand the behavior of the phenomena that generated the data.

Over the past decade, performance visualization has been closely tied to performance analysis of parallel systems. Visualization has been defined as a method of computing that allows performance data to be transformed into graphical form, thus allowing researchers to observe their simulations and measurements. Many visualization tools have been built for purposes such as program execution animation, data structure visualization, program debugging, and performance analysis. The Voyeur system [45] and its forerunner Balsa [13], and the Belvedere system [20] are all examples of systems that animate program execution. The Instant Display system [21] and Moviola [18] are examples of program debugging systems, while Hyperview [28], JED [26], IPS-2 [38] and Pablo [22] have been developed with performance analysis in mind.

While performance visualization can be done in real-time, i.e. visualizing the data while it is being generated, such systems are usually too complex to build and are usually limited by the speed of the display hardware. Most well-known visualization systems are *postmortem*: the performance data is collected from an execution run and analyzed after the program's execution. The performance data can be visualized at speeds controlled by the user. This overview is concerned only with postmortem systems.

Performance visualization consists of two stages: performance data collection and performance data visualization [44]. Performance data collection is concerned with generation of the data and its representation in a format that is easily accessible in the visualization phase. The visualization phase is concerned with generating meaningful displays of the data.

Data collection is performed by instrumenting a program and then running it. Instrumentation involves inserting probes at specific points in the program to extract information. Probes can be inserted at the hardware, operating system or application level. Hardware instrumentation usually involves having special purpose monitoring hardware within the computer system to perform non-intrusive performance data capture. While hardware instrumentation does provide accurate performance data, currently it is not widely used because a user is restricted to collecting performance data on a platform that provides the special-purpose hardware required. Examples of hardware instrumentation can be found in Hyperview [28, 27] and TMP [40]. Hyperview provides a instrumentation backplane for the Intel iPSC/2 Hypercube and TMP is a hardware instrumentation system used in a 680x0-based parallel architecture. Operating system instrumentation and application level instrumentation are examples of software instrumentation. They are widely used but tend to perturb the program execution and must be used judiciously. Pablo [22] and IPS-2 [38] are examples of performance analysis systems that have source code instrumentation subsystems.

The structure of the performance data varies from system to system, but typically, a performance data file (also known as a *trace file*) consists of a sequence of data records. Each record contains an *event*, which is information collected by an instrumentation probe, the time at which the event occurred, known as a timestamp and other relevant information pertaining to the event. Timing information is usually provided by a system clock. If the program is executed in parallel, then use of a system clock synchronized across multiple processors simplifies analysis.

The data display phase takes a performance data file and generates meaningful displays of some facet of the program's behavior as captured by the trace file. Examples of aspects of the program trace file that can be displayed are the utilization of processors, the time the program spent between any two specific events (e.g a procedure entry and the corresponding exit) and a display of the chronological ordering of trace file events. Examples of some displays include histograms of event occurrences, textual dumps of the trace file, X-Y plots, kiviat graphs, LEDs, contour plots, and scatter plots [22].

Earlier in this chapter, we suggested that the volume and complexity of performance data complicates the inspection of the data. Visualization effectively simplifies the process of assimilating data because it increases the rate at which information is presented to the user. However, allowing users to examine large quantities of data through histograms, time-lines, or Gantt charts does not provide anything more than a superficial analysis of the system under study. Our claim is that mere visualization does not imply a thorough analysis of data.

2.1.1 The Chitra Perspective on Visualization

In most existing performance analysis systems, visualization plays the major role – it is an end to itself. A parallel program performance analysis tool is summed up as an event tracing facility to capture trace information and a visualization front end that offers a variety of ways to display the collected trace data so as to lend insight into the relationships among the data. In contrast, the objective of the CHITRA92 system is to generate a parameterized, empirical model fitting a set of program execution traces and an input parameter set. The efficacy and accuracy of the parameterized model is established by the accuracy of the model's predictions on a different values of the input parameter set.

Visualization is of immense help in assisting a user to apply transformations on a program execution trace. Performance data is far more readily assimilated when presented graphically than when output as large volumes of textual data. For example, identifying recurrent patterns in a 2D X-Y time plot of an execution trace is easier done than while examining a text file of the trace data. Another example of where visualization is of great use is in clipping off the transient portion of the trace.

The CHITRA92 system uses visualization as a *means* to generating an empirical model of the behavior of the program. Chitra does not provide complex viewing systems which have no purpose other than displaying some facet of program output. Chitra does not provide a toolkit and expect that the user be left to assembling his/her performance analysis tools. What we do provide is a methodology for performance analysis and the Chitra performance analysis system is an organized collection of tools for realizing this methodology.

2.2 The IPS-2 System

IPS-2 [38, 37] is a performance measurement system being developed at the University of Wisconsin, Madison. IPS-2 has goals very similar to those of CHITRA92.

2.2.1 Objectives

IPS-2 is a true performance diagnosis tool. It takes a program execution trace of a parallel program and attempts to identify performance bottlenecks through two means of analysis: critical path analysis and phase behavior analysis. Critical path analysis identifies the longest path in the graph while phase behavior analysis sections the program into phases, so that critical path analysis can be applied to each phase. IPS-2 is unique in that it allows the programmer to link a performance bottleneck back to the source code responsible for the bottleneck. The IPS-2 performance diagnosis tool has versions for parallel computer systems, like the Sequent Symmetry, and for loosely coupled distributed computer systems (e.g. network of workstations).

2.2.2 Description

IPS-2 defines a computational hierarchy on the program being monitored. At the highest level is the program which can be thought of as a black box, having a set of inputs and generating a set of outputs. The next level is the machine level, where the program can be split into several concurrently executing processes on different processors (machines). The third level is the process level which represents a program as a collection of communicating processes. The fourth level is the procedure level where the distributed program is represented as a sequentially executed procedure call chain. The fifth and final level is the primitive activity level.

There are four basic components in IPS-2: instrumentation probes, the data pool, the analysts and a user interface. The instrumentation probes generate trace data when interesting events happen during program execution. The data pool stores trace data and caches intermediate results from the analyst. The analyst is set of concurrently executing processes that summarizes and evaluates the performance data. The user interface interacts with user and presents results.

A number of instrumentation techniques have been provided. Two of them are notable. The first is an enhanced gprof-style profiler which is used to record both procedure entry events as well as procedure exit events. The second is a modified run-time library. All instrumentation is done at the operating system level.

Each processor (parallel systems) or machine (distributed systems) contains a slave analyst that analyzes trace data from all processes on that machine. A master analyst process is responsible for collecting results from the slave analysts. The user interface to the system is a front-end to the master analyst.

2.3 JED

(J)ust another (E)vent (D)isplay [26] is a simple trace visualization tool that was developed at the University of Illinois, Urbana-Champaign. It has been included as one of the four systems being reviewed because its design and functionality are illustrative of design philosophies that are commonly found in most performance visualization tools.

2.3.1 Objectives

The objective of the JED project [26] was to design a simple event display tool that provided basic trace management support, user-definable event specification, and a user-customizable graphical presentation based on a standard Gantt chart display and a user-extensible analysis and display architecture.

2.3.2 Description

The JED system is targeted toward parallel, multitask programs running on the Cedar multiprocessor system. In this environment, parallel programs use the multitasking capabilities of the Xylem operating system to partition themselves into individual tasks for execution on the Cedar clusters. A task can further take advantage of hardware concurrency support of upto eight processors on each cluster, an Alliant FX/8, to execute loops in parallel.

The Cedar system has a built in performance data collection facility which is implemented as a library of counting, timing, and tracing routines that use a trace buffering run-time system for storing performance data. A trace file is provided for each executing task – the trace file is an time ordered merge of the task's eight processor trace streams.

The JED visualization system provides four separate components. A trace control component is responsible for reading the trace, positioning within the trace and searching for particular events. An event control component allows the association of names and graphic icons to events. A task display component opens viewports on the trace and allows events for tasks assigned to the viewports to be displayed in a Gantt chart-style form. Finally, the event display component controls how events are shown when "selected" in the task display. A standard event display is provided with the option to deselect it and select a user-defined event display.

JED implements efficient browsing to browse through large task event traces. These include a per task control, indexing of task trace files and event caching.

JED is easily customizable. There are no semantics associated with the event format. All event information is provided by the event definition file set up by the user. The user can control how events are shown in the task displays, and finally the user can replace the standard textual event display with a custom display.

2.4 MemSpy

While all of the other systems reviewed in this chapter are primarily execution trace analyzers, MemSpy [29] is different because it analyzes an application based on its memory reference behavior. MemSpy is a prototype tool developed at Stanford University which helps programmers identify and fix bottlenecks in both sequential and parallel programs. While most systems are involved in code oriented performance tuning, MemSpy is one of few tools that perform data oriented as well as code oriented tuning of programs.

2.4.1 Objectives

MemSpy's key philosophy is that while at the most basic level the memory reference behavior of an application depends on the intrinsic nature of an application, the programmer still has considerable flexibility in manipulating the algorithm, data structures, and program structure to change the memory reference structure in order to better exploit the memory hierarchy. MemSpy's primary goals are that it separately reports processor and memory time, so that programmers can discern where and when memory is the bottleneck, link the bottleneck back to specific data objects, and code sections, and finally give memory statistics at a level that allows the programmer to identify and fix the bottleneck.

2.4.2 Description

MemSpy is noted for the following features:

- It provides both data and code oriented output.
- It initially focuses attention on the bottleneck areas by providing the fraction of the time spent stalled on memory.
- It provides detailed information on the causes of poor memory performance.
- It is applicable to both parallel and serial applications.

MemSpy's first action is to rank each code and data object in the program according to the fraction of stall time they are responsible for. This metric is far more useful than others because code or data segments with high miss rates but low total stall time (because of few memory references) do not impact performance as much as segments with lower miss rates but higher total stall time. The initial output table generated by MemSpy presents a breakdown of memory stall time for each combination of code-segment and data object. For each combination, detailed information such as miss rate, read and write statistics, and statistics on local versus remote misses is generated. Finally a key feature is the breakdown on types of cache misses. MemSpy identifies three different kinds of cache misses:

1. if a line in the cache has never been referenced by the processor,
2. if the line has been replaced out of the cache since its last reference, or
3. if the line has been invalidated since its last reference.

For each type of cache miss, MemSpy provides statistics which breakdown the misses occurring due to each type of cache miss.

The trace information that is analyzed by MemSpy is available in two ways: simulation or hardware tracing. The current prototype uses simulation to gather data. Work is underway to provide a version that uses hardware tracing facilities.

2.5 PIE

PIE [30] is a software development environment for debugging performance that provides programmers ways to observe how computations execute by making use of special development and runtime visualization tools.

2.5.1 Objectives

The PIE system is a theoretical framework for developing techniques to predict, detect and avoid performance degradation in parallel and distributed programs. PIE is implemented on top of the Mach operating system. It supports languages such as C, Ada, Fortran, C-threads, and MPC.

2.5.2 Description

PIE provides a customized visual editing system which allows a user to identify the principal programming constructs within the program. After visualizing the source code, PIE allows the selection of constructs by the user for automatic observation within the program. All PIE's instrumentation is currently done using software instrumentation techniques. PIE then provides visualization utilities such as histograms and time-lines by which the performance data can be visualized. An interesting aspect of the PIE system is that it provides compensation algorithms by which the perturbation induced by the sensors and instrumentation probes can be minimized.

2.6 Mtool

Mtool [31] is an integrated tool for isolating performance bottlenecks in shared memory multiprocessor applications. Mtool uses low-overhead instrumentation methods and provides a variety of attention focusing mechanism for performance bottlenecks. The current Mtool implementation runs on MIPS-chip systems like DEC workstations, SGI multiprocessors and the Stanford DASH multiprocessor.

2.6.1 Objectives

The primary objective of Mtool is to characterize a program's execution based on the following execution time taxonomy:

1. Compute Time
2. Memory Overhead
3. Synchronization Overhead and
4. Extra Parallel Work

All of these categories of execution time analysis are supported for C programs (parallelized with the ANL macros) and Fortran programs (parallelized by an optimizing compiler).

2.6.2 Description

The Mtool analysis of a program's memory bottlenecks is done in two stages. In the first stage, Mtool instruments the program at the basic blocks. Based on the instruction latencies and a count of the number of times a basic block was executed, Mtool builds an execution profile of the program based on the amount of time it spent in each block. This profile is useful in identifying important regions of the program which need to be instrumented using probes and to ensure that the instrumentation is minimized to prevent overly perturbing the program's execution.

After the initial run of the program, Mtool instruments the program to collect two kinds of information: the time spent in selected loops, procedures and synchronization calls, and the basic block counts. Memory losses are isolated by comparing the actual time measurements to compute time estimates made using the basic block counts assuming an ideal memory system. Synchronization overheads are calculated directly from the execution time measurement of the synchronization calls. Finally, users can tag instructions as "extra work". The tool also automatically tags parallel control constructs as "extra work" within the analysis.

An window-based user interface displays information about bottlenecks hierarchically at a whole program, process, procedure, loop and synchronization object level. Mtool also provides the ability to link a memory overhead back to regions within the source code.

2.7 Pablo

Pablo [22] is a performance analysis environment being developed at the University of Illinois, Urbana-Champaign. Pablo is designed to provide performance data capture, analysis and presentation across several scalable parallel systems.

2.7.1 Objectives

As projects go, Pablo is extremely ambitious. The fundamental objectives of the system are to provide portability of the system across a variety of target platforms, to ensure that the performance analysis environment is scalable*, and thirdly, to provide an extensible performance analysis environment. The Pablo system classifies performance tool users as novices, intermediate users, and expert users. While the base system is sufficient for novice and intermediate users, expert users are expected not only to be able to reassemble existing toolkit components, but also add their own toolkit components to the system.

2.7.2 Description

Pablo is best described as a toolkit for the construction of performance analysis environments. It consists of a portable source code instrumentation subsystem and a performance data analysis subsystem with a trace data metaformat coupling the two subsystems. The portable source code instrumentation is designed to support interactive specification of source code instrumentation points. It consists of a graphical interface for source code specification, modified C and Fortran parsers that emit instrumented code and a library

*It is imperative that as the parallel system increases in size that the performance analysis environment should also scale commensurately.

of trace capture templates whose members can capture performance data generated by the instrumented source code when it actually executes on a parallel or distributed machine.

The performance analysis component of Pablo consists of a set of data transformation modules that can be graphically interconnected to form an acyclic directed, data analysis graph. Performance data flows through the graph nodes and is transformed to yield the desired performance metrics. Each performance data transformation module consists of a data transformation core (that performs the actual transform) and a data access "wrapper" that can read and write a self-documenting data stream meta-format (SDDF). The meta-format contains internal definitions of data types, sizes and names, but does not include embedded semantics.

All source code of the system has been written with portability as the primary objective. Also, because the data analysis modules communicate by message passing, individual modules can execute in parallel allowing the construction of scalable data analysis graphs.

An interesting aspect of the Pablo system is a sonification component by which performance data can be displayed by the use of sonic data presentation via a replay of sampled sounds and the use of a sound synthesizer.

2.8 Summary

All of the above systems have proven successful in different environments, and as such it is difficult to rate them against one another. We have identified the following criteria for comparison:

1. **Tools for gathering data**

   It is useful for any performance analysis system to provide tools to gather data. JED is targeted specifically towards parallel programs on the Cedar multiprocessor system and uses the Cedar performance measurement facility. JED, therefore, can read only a fixed data format. MemSpy is a prototype tool and can currently evaluate data provided by the Dash multiprocessor memory simulator. Mtool and PIE both provide different levels of source code and operating system instrumentation. Pablo provides a portable instrumentation software system which allows graphical insertion of software probes. Trace capture libraries are provided for the Intel Paragon and the CM-5. Pablo's Self Describing Data Format (SDDF) is also very useful for defining performance data formats. IPS-2 provides a software instrumentation library, and a modified version of gprof that collects both procedure entry as well as exit event data. CHITRA92 does not provide any tools for data capture, but provides a trace description language (CSL) very similar to Pablo's SDDF, which allows trace data to be both portable as well as efficient.

2. **Tools for analysis**

   Because trace data is usually very large (often megabytes in length), it is useful to provide analysis tools that assist a user in the dissemination of the data. JED provides a control mechanism by which the user can specify which events are to be displayed. MemSpy generates a variety of code and memory object statistics for memory cache misses in single and multi-processor environments. Mtool provides analysis of an application's memory profile based on a taxonomy of execution times. PIE is unique in that it provides compensation algorithms to minimize the effects of instrumentation sensors on perturbing the program. Pablo's analysis capabilities are wide-ranging but depend on the data analysis graph that is constructed by the user. IPS-2 provides two powerful analysis techniques that allow the user to locate bottlenecks in the program: critical path analysis and phase behavior analysis. CHITRA92 provides transformations for filtering, aggregation, clipping and allows user-defined transformations to be integrated into the system. It also provides spectral analysis techniques for identifying periodic behavior, and generates an emprirical model of the program.

3. **Sophistication of the interface**

   The sophistication of the interface is limited by the environment, e.g. the workstation capabilities and the system support.

JED is an event browser and provides a single Gantt chart-style display. MemSpy is really a tool that provides analysis and does not have a interface of significance. Mtool provides a simple window-based user interface. PIE is a visual programming environment and provides a visual editing system and displays such as histograms and time plots. Pablo has a wide range of graphical displays all which need to be connected in the data analysis graph. CHITRA92 provides X-Y views, text views, Fourier periodogram views, and empirical model views. Pablo and CHITRA92 both provide multiple-view visualization systems. IPS-2 provides a graphic editor which allows the programmer to modify the computational structure of the program being measured through its program hierarchy. It also provides a flexible user interface that allows the user to display performance metrics in graphical or tabular form or use the automatic guidance techniques.

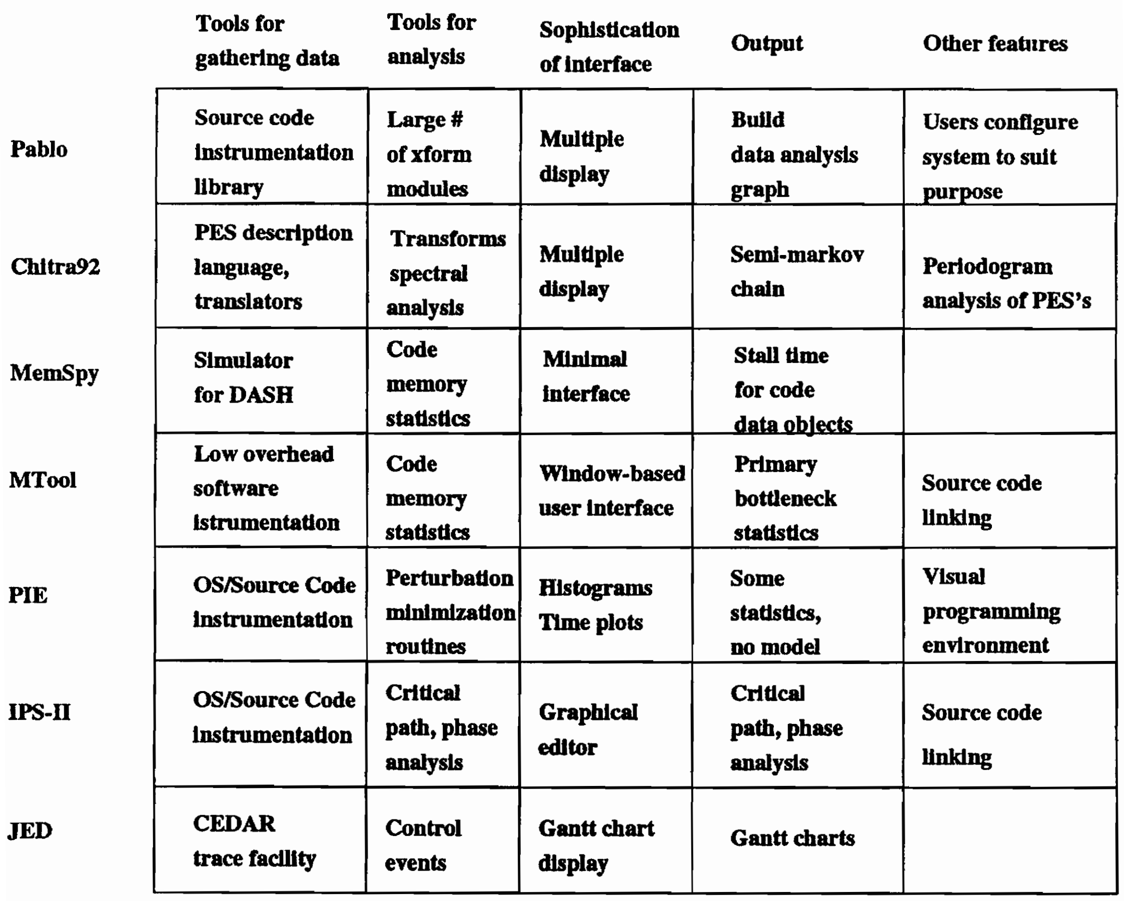
4. **Level of information presented**

The performance analysis tool can present information at different levels. The simplest level of information that can be presented is through basic graphics displays and listing of raw data. An intermediate level displays trace data in terms of the source program. Symbol tables, call graphs, flow graphs and data dependence graphs are all examples of intermediate level information presentation. The highest level of presentation is in terms of a high-level model of program behavior.

Of the six systems reviewed, JED offers merely a Gantt chart-style display of performance data. MemSpy qualifies as an intermediate level display because it generates information on the influence of a program's data objects and code objects on the memory reference pattern of the program. IPS-2 scores high in this category because it not only generates a high-level model of the program behavior using critical path analysis and phase analysis, it also links program bottlenecks back to the offending source code. PIE also provides source code links from the bottleneck, but has no high-level model. A lot of the analysis is left to the intuition of the user. Mtool attempts to perform a comparison between the performance of the program on an ideal memory system and the performance of an actual execution. The performance bottleneck statistics are only at a high level, though it does have the ability to display these statistics in a hierarchical manner based on the program's structure. Pablo does provide extremely powerful display interfaces and the ability to analyze a wide range of data formats. Its biggest problem is that it does not support any kind of performance analysis methodology. The user has to provides his/her own methodology to generate a high-level model. CHITRA92 is another example of a system whose focus is on generating a high-level model of the program using a semi-Markov chain.[†].

Figure 2.1 is a table which summarizes our comparision of these six system with CHITRA92.

[†]Source code linking is being planned in future versions of the system.



	Tools for gathering data	Tools for analysis	Sophistication of interface	Output	Other features
Pablo	Source code instrumentation library	Large # of xform modules	Multiple display	Build data analysis graph	Users configure system to suit purpose
Chitra92	PES description language, translators	Transforms spectral analysis	Multiple display	Semi-markov chain	Periodogram analysis of PES's
MemSpy	Simulator for DASH	Code memory statistics	Minimal interface	Stall time for code data objects	
MTool	Low overhead software istrumentation	Code memory statistics	Window-based user interface	Primary bottleneck statistics	Source code linking
PIE	OS/Source Code instrumentation	Perturbation minimization routines	Histograms Time plots	Some statistics, no model	Visual programming environment
IPS-II	OS/Source Code instrumentation	Critical path, phase analysis	Graphical editor	Critical path, phase analysis	Source code linking
JED	CEDAR trace facility	Control events	Gantt chart display	Gantt charts	

Figure 2.1: Table comparing other performance analysis tools with CHITRA92.

Chapter 3

Methodology

This chapter introduces and explains several key concepts that are unique to the CHITRA92 performance analysis system. In doing so, we also present the methodology used to model a parallel or distributed program.

We first discuss in detail the notion of a *program execution sequence* (PES), which is fundamental to CHITRA92 performance analysis system. We then describe semi-Markov chain model of a PES and our reasons for doing so. Next, we describe four transforms which serve as powerful tools for reducing a PES to a form that generates a simpler model. We also discuss techniques for identifying periodicity in a PES, and finally we propose steps that should be followed to generate a semi-Markov model of a PES.

3.1 Program Execution Sequences (PES's)

Consider a sequential program executing on a uniprocessor architecture. We can associate with each program a function from the input domain to the output domain. To do so, we formalize the notion of state (the concept of memory (that binds identifiers to values), input and output). For simplicity, we assume that the program deals only with integer values and booleans which may result from boolean expressions. Symbol Z stands for the set of integers and the symbol $\perp$ stands for the undefined value.

For any given sequential program P, P's state s_P is formalized by a triple

$$s_P \quad = \quad \langle mem_P, i_P, o_P \rangle,$$

where:

- mem_P is a function that gives the value of each program variable (if Id_P denotes the set of P's variables, then $mem_P : Id_P \rightarrow Z \cup \perp$).
- i_P and o_P are the input streams and output streams respectively.

Let us associate with our sequential program a set of data items called *program parameters*, whose values can be modified every time the program is executed. Examples of program parameters are program constants and input data items to the program. We will denote the set of program parameters as $\{x_0, \ldots, x_i, \ldots, x_k\}$, where x_i is a program parameter whose value can be modified, for $0 \leq i \leq k$.

Any sequential program execution can therefore be modeled by a sequence of states and a set of program parameters. Let P_E denote an execution of the sequential program P. Then

$$P_E \;=\; ((s_0, s_1, s_2, \ldots, s_n), \{x_0, x_1, x_2, \ldots, x_n\}),$$

where $(s_0, s_1, s_2, \ldots, s_n)$ represent the state sequence of P and $\{x_0, x_1, x_2, \ldots, x_n\}$ represents the set of input parameters to P_E. A program instruction is therefore simply a transition function that maps one program's state to the next state in the sequence. In a sequential program, the given input parameter set uniquely determines the state sequence.

We can extend this notion of state from sequential to parallel programs as well. Let us assume that our parallel programs execute on a MIMD architecture. In our model of parallel programs, a parallel program is a program where multiple threads (or processes) interact, competing for and sharing resources. A thread (or a process) is a code fragment scheduled by the underlying operating system. Instead of a triple to define state as in a sequential program, we have an n-tuple which provides an instantaneous description of the program. This n-tuple represents the program state of the executing program. Each component in the program state is simply a state s_{P_i} for each of the n executing threads in the program $0 \leq i \leq n-1$.

For a parallel program P, its *program state* s_P is an n-tuple providing an instantaneous description of the program. We have

$$s_P = \langle s_{P0}, s_{P1}, s_{P2}, \ldots, s_{P(n-1)} \rangle$$

where

- s_P is the program state of the parallel program.
- s_{Pi} where $0 \leq i \leq (n-1)$ represents the state of thread i at some time instant. We can further expand $s_{Pi} = \langle mem_{Pi}, i_{Pi}, o_{Pi} \rangle$ where mem_{Pi} is a function that gives the value of each program identifier for thread i, i_{Pi} and o_{Pi} are the input streams and output streams respectively for thread i.

The model of a parallel program's execution is not as straightforward as the model for a sequential program. A parallel program's execution can be represented by one of *many* state sequences for the same set of input parameters. Each state sequence represents one possible execution of the program. The possibility of multiple execution sequences exists because parallel programs are inherently non-deterministic in their behavior. Let P_E denote an execution of P. Then we have:

$$P_E = (\{R_0, R_1, R_2, R_3, \ldots, R_M\}, \{x_0, x_1, x_2, \ldots, x_n\})$$

where

- $\{R_0, R_1, R_2, \ldots, R_M\}$ represent the set of program state sequences. P_E will choose some program state sequence R_i where $0 \leq i \leq M$. The set of program state sequences illustrates the nondeterministic nature of P's execution.
- $\{x_0, x_1, x_2, ..., x_n\}$ represents the set of input parameters to P.

In practice, the program state of a program is the instantaneous description of the program. For any program, this information might consists of hundreds of memory variables (local or global), several input and output streams, and code pointers to the next instruction to be executed in each of the concurrent threads.

Under ideal circumstances, an instrumentation system could capture every program state of the program for a given input parameter set and execution trajectory to answer all performance questions. However because the state space could be very large, we specify events in the program that are of interest to us and generate a subset of the program state space. The program state is therefore an n-tuple where each component represents some information relevant to the performance study. For each program execution, and input parameter set, we can generate a *program execution sequence* (PES), which is a sequence of ordered pairs – each ordered pair consisting of a program state s_i and an entry time e_i which represents the time the execution trajectory entered program state s_i. We can map a PES into two constituent sequences: a *program state sequence* (also called a *symbol sequence*) and an *occupancy time sequence*. The program state sequence consists of the sequence of all program states in the PES, and the occupancy time sequence is a sequence of the occupancy times of the states in the program state sequence.

Consider the PES

$$I \quad = \quad (s_0, e_0), (s_1, e_1), (s_2, e_2), \ldots, (s_m, e_m).$$

The symbol sequence corresponding to I is

$$I_s = s_0, s_1, s_2, \ldots, s_{m-1}$$

and its program occupancy time sequence is

$$I_o = (e_1 - e_0), (e_2 - e_1), (e_3 - e_2), \ldots (e_m - e_{m-1}).$$

Note that the last state s_m is not present in the program state sequence. This is because its occupancy time cannot be computed. The two sequences, I_s and I_o, are equal in length.

Example 3.1.1 *Consider Dijkstra's "Dining Philosophers" problem [4]. The problem can be stated as follows:*

N philosophers spend their lives thinking and eating. The philosophers share a common dining room where there is a circular table surrounded by N chairs, each belonging to a philosopher. In the center of the table is a large bowl of spaghetti, and the table is laid with N forks. Whenever a philosopher is hungry, he enters the dining rooom, sits on his own chair, and picks up the fork on the left of his place and on the right of his place to eat. When he has finished, he puts down both forks and leaves the room. Therefore, a philosopher can eat only when neither of his neighbors is eating. The problem is to write the algorithm for $Philosopher_i$ $0 \leq i \leq N-1$.

The implementation of the dining philosophers problem using threads (Two examples of threads packages are Cthreads, a threads package for distributed programming that is provided with the Mach operating system [33], and Presto, a C++ threads package for parallel programming [11]) is straightforward. Each philosopher is represented as thread which is forked off by the parent task and then detached. Since forks are shared resources that adjacent philosophers can access, they must be protected using mutual exclusion variables such as semaphores.

The solution outline is as follows:

- *A philosopher i is represented as a thread. It performs in an always-true loop, the thinking, fork acquiring, eating and fork releasing functions.*

- *A fork is represented as a semaphore. Accessing a fork corresponds to locking the semaphore. Releasing a fork corresponds to unlocking the semaphore.*

Figure 3.1 shows a code fragment for a single thread representing a philosopher for $N = 5$. *The following symbols represent instrumentation points within a thread: A1 implies philosopher i acquiring the ith fork, A2 implies acquiring the* $i+1$*th fork, E implies eating, R1 implies releasing the ith fork, R2 implies releasing the* $i+1$*th fork, and T implies thinking.*

For N = 2, a sample PES for the above problem, I is as follows:

$$I = (A2T, 1550), (ET, 1560), (EA1, 1570), (R1A1, 1585), (R2A1, 1597), (R2A2, 1601),$$

```
#define MAXPHIL 5
#define TRUE 1

semaphore forks[MAXPHIL];

void philosopher(int i)
{
        while (TRUE)
        {
                think(); /* Think (T) */
                sem_lock(forks[i%MAXPHIL]); /* Acquire ith fork (A1)*/
                sem_lock(forks[(i+1)%MAXPHIL]); /* Acquire (i+1)th fork (A2) */
                eat(); /* Eat - (E) */
                sem_unlock(forks[i%MAXPHIL]); /* Release ith fork (R1) */
                sem_unlock(forks[(i+1)%MAXPHIL]); /* Release (i+1)th fork (R2) */
        }
}
```

Figure 3.1: Code fragment for a single philosopher in the Dining Philosophers problem.

$(TA2, 1607), (TE, 1621), (A1E, 1633), (A1R1, 1651), (A2R1, 1668), (A2R2, 1669),$
$(A2T, 1685), (ET, 1686), (EA1, 1695), (R1A1, 1711), (R2A2, 1723), (TA2, 1731),$
$(TE, 1735), (A1E, 1745), (A1R1, 1760)$

We can split I into its two constituent sequences.

The program state sequence is

$$\begin{aligned} I_s &= (A2T, ET, EA1, R1A1, R2A1, R2A2, TA2, TE, A1E, A1R1, A2R1, \\ &\quad A2R2, A2T, ET, EA1, R1A1, R2A2, TA2, TE, A1E, A1R1) \end{aligned}$$

and the program state occupancy time sequence is

$$I_o = (10, 10, 15, 12, 4, 6, 14, 12, 18, 17, 1, 16, 1, 9, 16, 12, 8, 4, 10, 15)$$

Figure 3.2 is a time plot of PES I, Figure 3.3 is the semi-Markov chain model of I.

3.2 Semi-Markov Chains

CHITRA92 uses semi-Markov chains as its model of program behavior. In this section we briefly review the theory of semi-Markov processes. For more information, the reader is referred to the literature [42, 12].

A *stochastic process* can be defined as a set of random variables $X(t)$, where the random variables are indexed by the time parameter t. To completely describe a stochastic process three issues must be addressed: the *state space*, the *index (time) parameter*, and the *statistical dependencies* among the random variables for different values of the time parameter t.

The state space refers to the set of possible values (states) that $X(t)$ may take. If the set of states is finite or countable, then we say that the process is a *discrete-state* process or a *chain*. On the other hand, if the set of states are over a finite or infinite continuous interval, then the process is a *continuous-state* process.

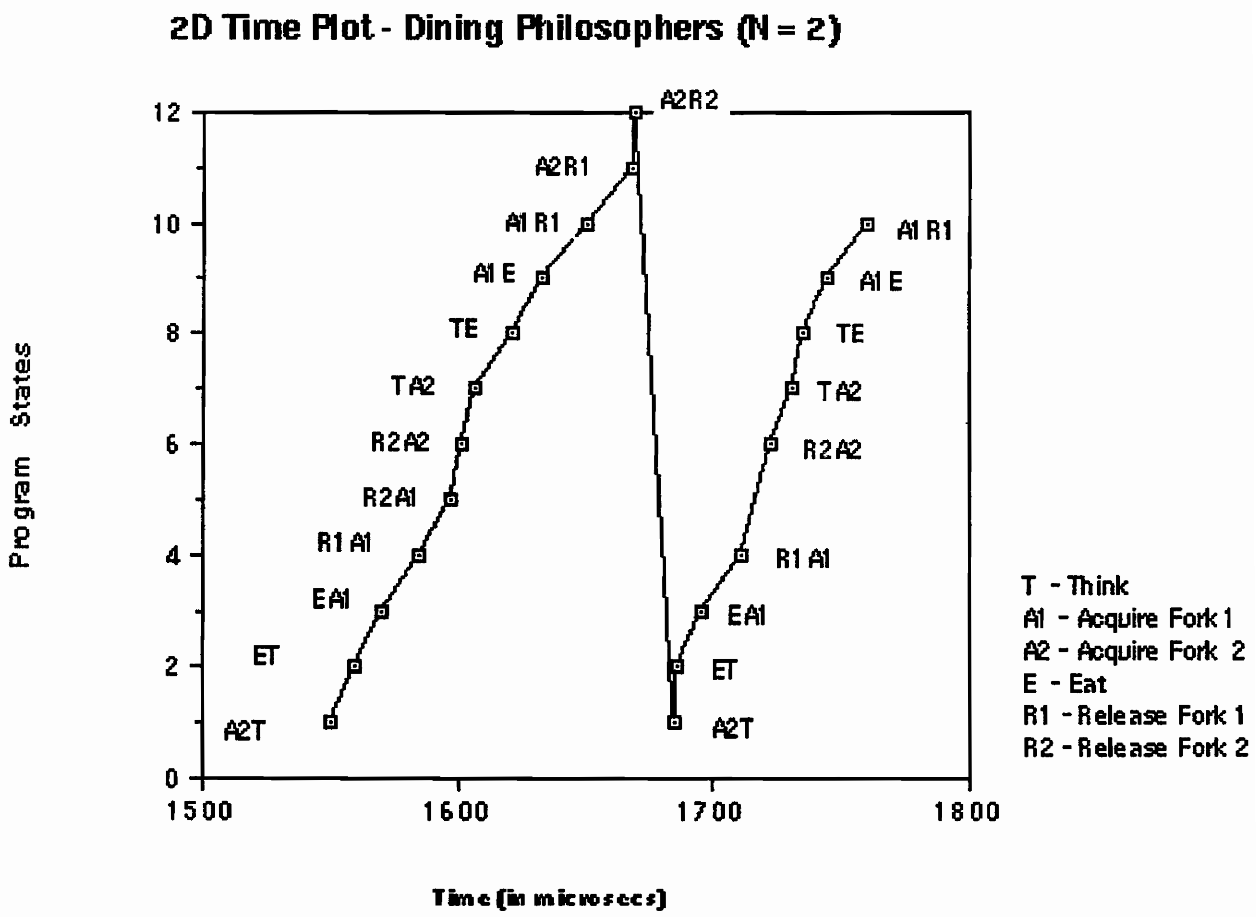


Figure 3.2: 2D Time Plot for a 2 philosopher Dining Philosophers problem.

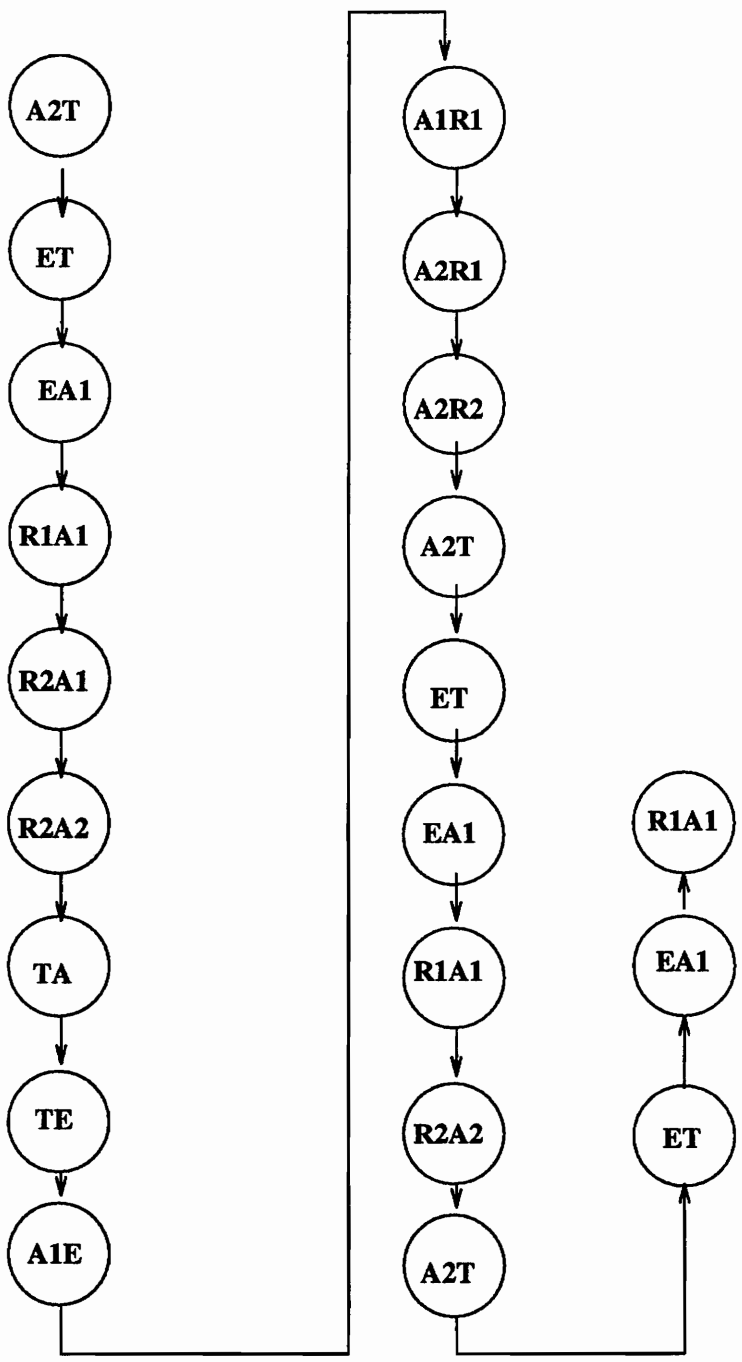


Figure 3.3: Semi-Markov chain model for a 2 philosopher Dining Philosophers problem.

The second issue is the index (time) parameter. If the permitted times at which transitions can be made from one state to another are finite or countable then the stochastic process is called a discrete-(time) process. If these transitions can be made at any finite interval in time, then the stochastic process is referred to as a continuous-parameter process. A discrete-time stochastic process is often referred to as a stochastic sequence and the random variable is denoted by X_n rather than $X(t)$.

The final issue which serves to truly distinguish between stochastic processes is the relationship between the random variables that constitute the stochastic process. There are several types of stochastic processes that are characterized by different kinds of dependency relationships among their random variables. Examples are stationary processes, independent processes, Markov processes, birth-death processes, homogeneous processes and non-homogeneous processes, semi-Markov processes, and renewal processes. We restrict our discussion to Markov and semi-Markov processes.

A Markov process with a discrete state space is referred to as a Markov chain. A set of random variables forms a Markov chain if the probability of the next state being in some state x_{n+1} depends only on the current value of the state x_n, and not on any previous values. Thus the random sequence has a dependency which extends backwards one unit in time. The same dependency holds for continuous-time Markov chain except that the transitions between states may take place at any time instant, whereas for a discrete-time Markov chain the time instants at which state changes may occur are assigned integers $0, 1, 2, \ldots, n$. Also, a homogeneous Markov chain is one for which the transition probabilities are independent of time.

Because the Markovian property requires that the past history be completely summarized by the current state, it imposes a strict restriction on the distribution of time that a process may remain in a particular state. For continuous time Markov chains the occupancy time can only be exponentially distributed, while for discrete-time Markov chains, the occupancy time can only geometrically distributed.

We can relax that restriction by permitting an arbitrary distribution of time that the process may remain in a state. A discrete-time (or continuous-time) Markov chain where an arbitrary state occupancy time distribution is permitted is called a *semi-Markov chain.* However the condition of computing transition probabilities only from the current state still holds. At the instants when a transition occurs, a semi-Markov chain behaves like an ordinary Markov chain.

3.2.1 Why use homogeneous continuous-time semi-Markov chains?

To model a PES, it is necessary to map every program state to a model state. The model requires a transition function to map the current model state to a subsequent model state. In Chapter 1, we argued that parallel programs are inherently nondeterministic. Nondeterminism in a parallel program can correspond to random events such as process contention for resources or context switches. However, there are sections of the program that are deterministic as well. The execution of a sequence of program statements is one such example.

The choice of model, therefore, hinges on its ability to model nondeterministic behavior as well as deterministic behavior. The ideal model will be able to model deterministic as well as random behavior equally by combining deterministic as well as probabilistic transition functions [43, 19].

Chitra uses as its program behavior model a homogeneous, continuous time semi-Markov chain. Semi-Markov processes are appropriate because they model general state occupancy time distributions and therefore, are quite flexible. The strong Markov requirement that state transitions rates be determined solely on the basis of the current state is acceptable in theory.

3.3 Transformations

The representation of a program as a PES is the first step to constructing a semi-Markov chain model of the program. While it is possible to generate a model from the PES, such a model is not very useful because it rarely produces an execution trajectory that closely matches the input PES. To produce a satisfactory model of a PES, it is necessary to first *transform* the PES.

A *transformation* maps an input PES I to an output PES O. More precisely, a transform maps an input symbol sequence and an input occupancy time sequence, denoted I_s and I_o, to an output symbol sequence and an output occupancy time sequence, denoted O_s and 0_o, respectively. The transformation function may or may not take as input other parameters besides the PES itself. The output of a PES transformation is always another PES. A composition of transformations may therefore be applied on an input PES.

Applying transforms is an iterative and exploratory process. Sometimes the application of a sequence of transforms results in an unsatisfactory model being generated. The effects of the transform then need to be undone and an alternate set of transforms should be applied.

A generic transformation for a PES can be written as follows:

$$O := \Lambda(\{p_0, p_1, p_2, \ldots, p_n\}, I)$$

where

- Λ is a transformation function on the input PES,
- I is an input PES and O is an output PES, and
- $\{p_0, p_1, p_2, \ldots, p_n\}$ are an optional set of parameters to the transformation function.

We describe four different types of transformations used in Chitra which satisfy the above criteria. They are filtering, aggregation, clipping and projections. They are described in sections 3.3.1, 3.3.2, 3.3.3 and 3.3.4 respectively.

3.3.1 Filtering

Most programs exhibit some form of random behavior. Examples of occurrences of random behavior in executing programs are resource contention, context switches and page faults. The state of the program is no longer deterministic. Random behavior in a program manifests itself as sequences of program states that occur rarely.

Filters are transforms that can be used to identify random behavior in a PES by locating these infrequently occurring sequences. In most PES's, the program spends most of its execution time in a few states, termed *dominant states.* The program periodically passes through these dominant states, but between these states the program makes transitions through dissimilar subsequences that usually occur randomly. A dominant state that preceeds such a randomly occurring subsequence in the symbol sequence of the PES is called the *predecessor state* and the dominant state that succeeds the subsequence is termed the *successor state.*

Note that two (or more) such dissimilar subsequences can have the same predecessor and successor states. Each sequence represents the path taken by the program execution sequence as it moves from the predecessor state to the successor state. In terms of occupancy time spent in these states, the duration is small in comparison with the total duration of the PES, and in terms of the number of occurrences of these states, they are few in number.

The purpose of a filter transform is to replace these randomly occurring subsequences by a composite state, the *filter aggregate state.* Every subsequence having the same predecessor state and the successor state is replaced with the same filter aggregate state.

Consider the PES

$$I = (s_0, e_0), (s_1, e_1, (s_2, e_2), \ldots, (s_m, e_m).$$

Its program state sequence is

$$I_s = s_0, s_1, s_2, \ldots, s_{m-1},$$

and its program occupancy time sequence is

$$I_o = (e_1 - e_0), (e_2 - e_1), (e_3 - e_2), \ldots (e_m - e_{m-1}).$$

Time domain filtering

Let the sum of all times in the occupancy time sequence be T. For each program state s, let $h(s)$ denote the occupancy time of s in the PES. Let $t_{threshold}$ be the threshold occupancy time such for each state s, if $h(s)$ is less than $t_{threshold}$, then every instance of s in I is *filtered* out of I. The filtering process is defined as follows

- **Run identification:** Every subsequence of filtered out states in I constitutes a *run* in the PES. Let there be n runs after applying the time-domain filter on I: $(R_1, R_2, \ldots, R_n)$. For each run R_i, let the predecessor state be given by $Pred(R_i)$, and the successor state be given by $Succ(R_i)$. Let the null program state be denoted by ϕ, i.e. if R_i has no predecessor state, then $Pred(R_i) = \phi$ or if it has no successor state, then $Succ(R_i) = \phi$.

- **Run replacement:** All runs that have the same predecessor and successor states are replaced by the same newly generated filter aggregate state in the transformed output PES. The replacement is performed until all runs in the PES are eliminated.

Example 3.3.1 *Recall the PES of Example 3.1.1.*

$$\begin{aligned} I = {} & (A2T, 1550), (ET, 1560), (EA1, 1570), (R1A1, 1585), (R2A1, 1597), (R2A2, 1601), \\ & (TA2, 1607), (TE, 1621), (A1E, 1633), (A1R1, 1651), (A2R1, 1668), (A2R2, 1669), \\ & (A2T, 1685), (ET, 1686), (EA1, 1695), (R1A1, 1711), (R2A2, 1723), (TA2, 1731), \\ & (TE, 1735), (A1E, 1745), (A1R1, 1760). \end{aligned}$$

Its program state sequence is

$$\begin{aligned} I_s = {} & (A2T, ET, EA1, R1A1, R2A1, R2A2, TA2, TE, A1E, A1R1, A2R1, \\ & A2R2, A2T, ET, EA1, R1A1, R2A2, TA2, TE, A1E, A1R1), \end{aligned}$$

and the program state occupancy time sequence is

$$I_o = (10, 10, 15, 12, 4, 6, 14, 12, 18, 17, 1, 16, 1, 9, 16, 12, 8, 4, 10, 15).$$

The $h(s)/T$ value for $A1E$ is .147 and the $h(s)/T$ value for $EA1$ is .157. The occupancy time fraction is less than .147 for all other states. If we set a $t_{threshold}$ value such that $t_{threshold}/T = 0.147$ then all other states will be filtered out of I if we apply a time domain filter with $t_threshold$ on I. We can identify four runs, three of which are distinct. They are

1. $R_1 = \langle A2T, ET \rangle, Pred(R_1) = \phi, Succ(R_1) = EA1$
2. $R_2 = \langle R1A1, R2A1, R2A2, TA2, TE \rangle, Pred(R_2) = EA1, Succ(R_2) = A1E$
3. $R_3 = \langle R1A1, R2A2, TA2, TE \rangle, Pred(R_3) = EA1, Succ(R_3) = A1E$
4. $R_4 = \langle A1R1, A2R1, A2R2, A2T, ET \rangle, Pred(R_4) = A1E, Succ(R_4) = EA1$

Notice that runs R_2 and R_3 have the same predecessor and successor states. Let $T1$ be the filter aggregate state that replaces run R_1. Let $T2$ replace runs R_2 and R_3 and let $T3$ replace run R_4. The resulting output PES is

$$O = (T1, 1550), (EA1, 1570), (T2, 1585), (A1E, 1633), (T3, 1651), (EA1, 1695),$$
$$(T2, 1711), (A1E, 1745), (A1R1, 1760).$$

Its program state sequence is

$$O_s = (T1, EA1, T2, A1E, T3, EA1, T2, A1E, A1R1),$$

and the program state occupancy time sequence is

$$O_o = (20, 15, 48, 18, 44, 16, 34, 15).$$

Figure 3.4 is a time plot of the PES, O, after applying the above time domain filter on I and Figure 3.5 is the semi-Markov chain model of O.

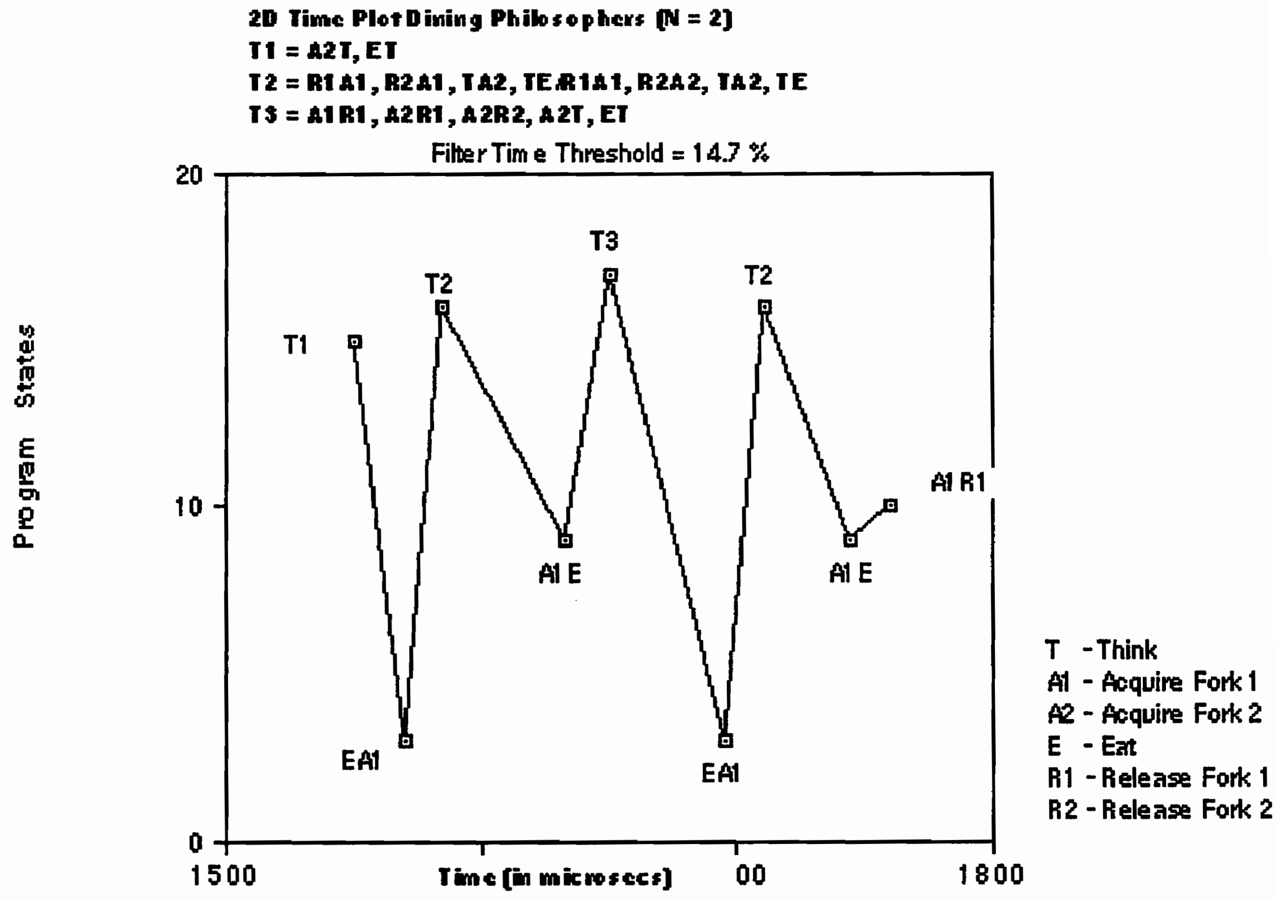


Figure 3.4: 2D Time Plot 2 philosopher Dining Philosophers problem – after time domain filtering.

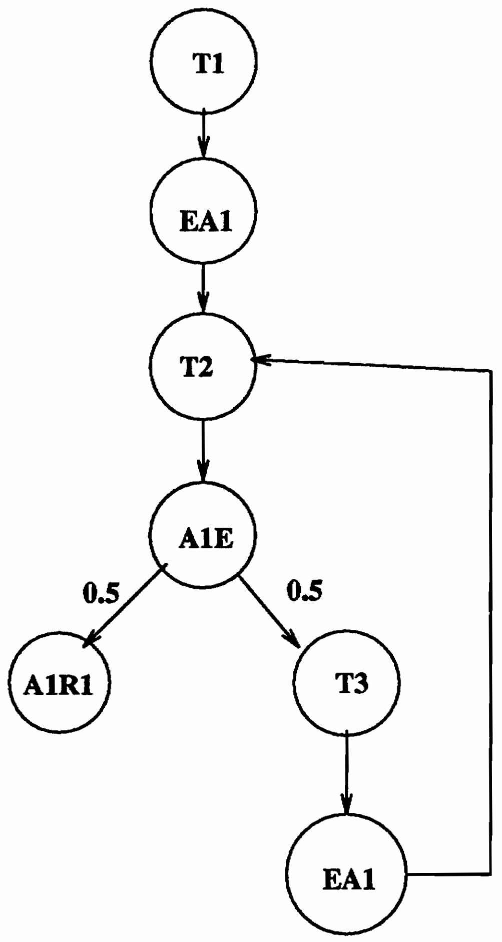


Figure 3.5: Semi-Markov chain model of a 2 philosopher Dining Philosophers problem – after time domain filtering.

Event domain filtering

Let the number of occurrences of each program state s, be given by $n(s)$. Let $n_{threshold}$ be the threshold number of occurrences such that for each state s if $n(s)$ is less than $n_{threshold}$, then every instance of s in I of filtered out of I. The filtering process is the same as that of time-domain filtering.

Example 3.3.2 *Now consider the PES in Example 3.1.1. If we apply an event domain filter, where $n_{threshold} = 2$, we get two new runs:*

1. $R_1 = \langle T1 \rangle, Pred(R_1) = (-1), Succ(R_1) = EA1$, *and*
2. $R_2 = \langle T3, TE \rangle, Pred(R_2) = A1E, Succ(R_2) = EA1$.

Let $T4$ replace run R_1 and let $T5$ replace run R_2. The resulting output PES is

$$\begin{aligned} O &= (T4,1550),(EA1,1570),(T2,1585),(A1E,1633),(T5,1651),(EA1,1695),\\ &\quad (T2,1711),(A1E,1745),(A1R1,1760). \end{aligned}$$

Its program state sequence is

$$O_s = (T4, EA1, T2, A1E, T5, EA1, T2, A1E, A1R1)$$

and the program state occupancy time sequence is

$$O_o = (20, 15, 48, 18, 44, 16, 34, 15).$$

Figure 3.6 is a time plot of the PES, O, after applying the above event domain filter on I.

3.3.2 Aggregation

While filters are useful for identifying random behavior in a PES, aggregation is useful in identfying deterministic behavior in a PES. Recall that a dominant state is a state which either occurs very often in a PES or is a state which has a very large occupancy time.

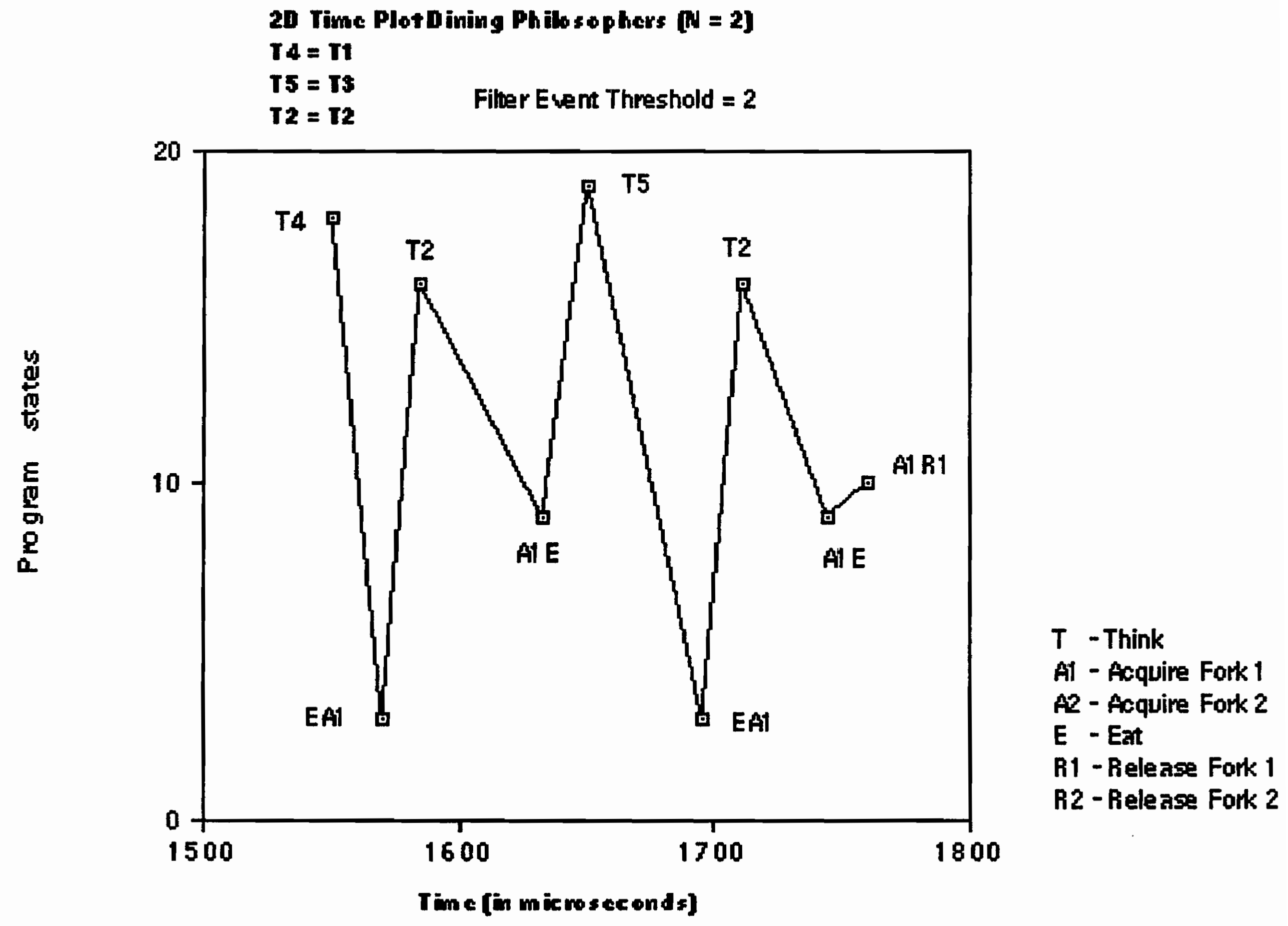


Figure 3.6: 2D Time Plot 2 philosopher Dining Philosophers problem – after event domain filtering.

The aggregation transform is used to identify and combine subsequences of dominant states within a PES. Once a dominant state subsequence has been identified, the aggregation transform replace all such identical subsequences with a new composite state, the *pattern aggregate state.*

Notice that the filter transform and the aggregation transform have complementary roles. If a PES is predominantly random in behavior, then successive filter transforms can be applied on the PES before generating the empirical model of the PES and if the PES shows predominantly deterministic behavior, then all deterministic subsequences can be combined by applying successive aggregation transforms.

Consider the PES

$$I = (s_0, e_0), (s_1, e_1), (s_2, e_2), \ldots, (s_m, e_m).$$

Its program state sequence is

$$I_s = s_0, s_1, s_2, \ldots, s_{m-1},$$

and its program occupancy time sequence is

$$I_o = (e_1 - e_0), (e_2 - e_1), (e_3 - e_2), \ldots (e_m - e_{m-1}).$$

Let us define a subsequence A such that

$$A = (a_0, a_1, a_2, \ldots, a_n),$$

where each a_i in A is a program state for all $0 \leq i \leq n$. The aggregation transform scans I for every occurrence of the pattern A in I. The sequence A is then replaced by a new pattern aggregate state denoted s_A. This is repeated for every occurrence of A in I. Every occurrence of s_A is assigned the timestamp associated with the first state in the subsequence that matches A.

Example 3.3.3 *Recall the PES of Example 3.1.1.*

$$\begin{aligned} I \;=\;& (A2T,1550),(ET,1560),(EA1,1570),(R1A1,1585),(R2A1,1597),(R2A2,1601),\\ & (TA2,1607),(TE,1621),(A1E,1633),(A1R1,1651),(A2R1,1668),(A2R2,1669),\\ & (A2T,1685),(ET,1686),(EA1,1695),(R1A1,1711),(R2A2,1723),(TA2,1731),\\ & (TE,1735),(A1E,1745),(A1R1,1760) \end{aligned}$$

Its program state sequence is

$$\begin{aligned} I_s \;=\;& (A2T,ET,EA1,R1A1,R2A1,R2A2,TA2,TE,A1E,A1R1,A2R1,\\ & A2R2,A2T,ET,EA1,R1A1,R2A2,TA2,TE,A1E,A1R1) \end{aligned}$$

and the program state occupancy time sequence is

$$I_o = (10,10,15,12,4,6,14,12,18,17,1,16,1,9,16,12,8,4,10,15).$$

Let us define an aggregation subsequence

$$A = R2A2,TA2,TE.$$

Applying an aggregation transform with A yields the output PES

$$\begin{aligned} O \;=\;& (A2T,1550),(ET,1560),(EA1,1570),(R1A1,1585),(R2A1,1597),\\ & (Z,1601),(A1E,1633),(A1R1,1651),(A2R1,1668),(A2R2,1669),\\ & (A2T,1685),(ET,1686),(EA1,1695),(R1A1,1711),(Z,1723),\\ & (A1E,1745),(A1R1,1760), \end{aligned}$$

where

$$Z = (R2A2,TA2,TE).$$

O's program state sequence is

$$\begin{aligned} I_s \;=\;& (A2T,ET,EA1,R1A1,R2A1,Z,A1E,A1R1,A2R1,\\ & A2R2,A2T,ET,EA1,R1A1,Z,A1E,A1R1) \end{aligned}$$

and the program state occupancy time sequence is

$$O_s = (10, 10, 15, 12, 4, 32, 18, 17, 1, 16, 1, 9, 16, 12, 8, 4, 10, 15).$$

Figure 3.7 is a time plot of the PES, O, after applying the above aggregation transform on I and Figure 3.8 is the semi-Markov chain model of O.

3.3.3 Clipping

The clipping transform deletes an initial and a final portion of the symbol sequence. The corresponding occupancy times are deleted from the occupancy time sequence. The clipping transform is useful in eliminating the initial and final (transient) portions of a PES, during which threads are being initiated and terminated. Clipping is also useful when program execution consists of several phases, because a different homogeneous chain model can be generated for each phase.

Clipping (event domain) Consider a PES I. Let n_i and n_f denote the ith and the fth program state in I_s. Then an event domain transform will remove all ordered pairs before the ith program state and all events after the fth program state, resulting in an output PES, O.

Clipping (time domain) Consider a PES I. Let t_i and t_f be any two time stamps. Then a time domain clipping transform will remove all ordered pairs that have an entry time stamp value less than t_i or greater that t_f resulting in an output PES, O.

Example 3.3.4 *Recall the PES I from Example 3.1.1. Applying an event mode clipping transform with $n_i = n_f = 8$ to I_s, we generate an output PES*

$$O = (A1E, 1633), (A1R1, 1651), (A2R1, 1668), (A2R2, 1669).$$

Its program state sequence is

$$I_s = (A1E, A1R1, A2R1, A2R2)$$

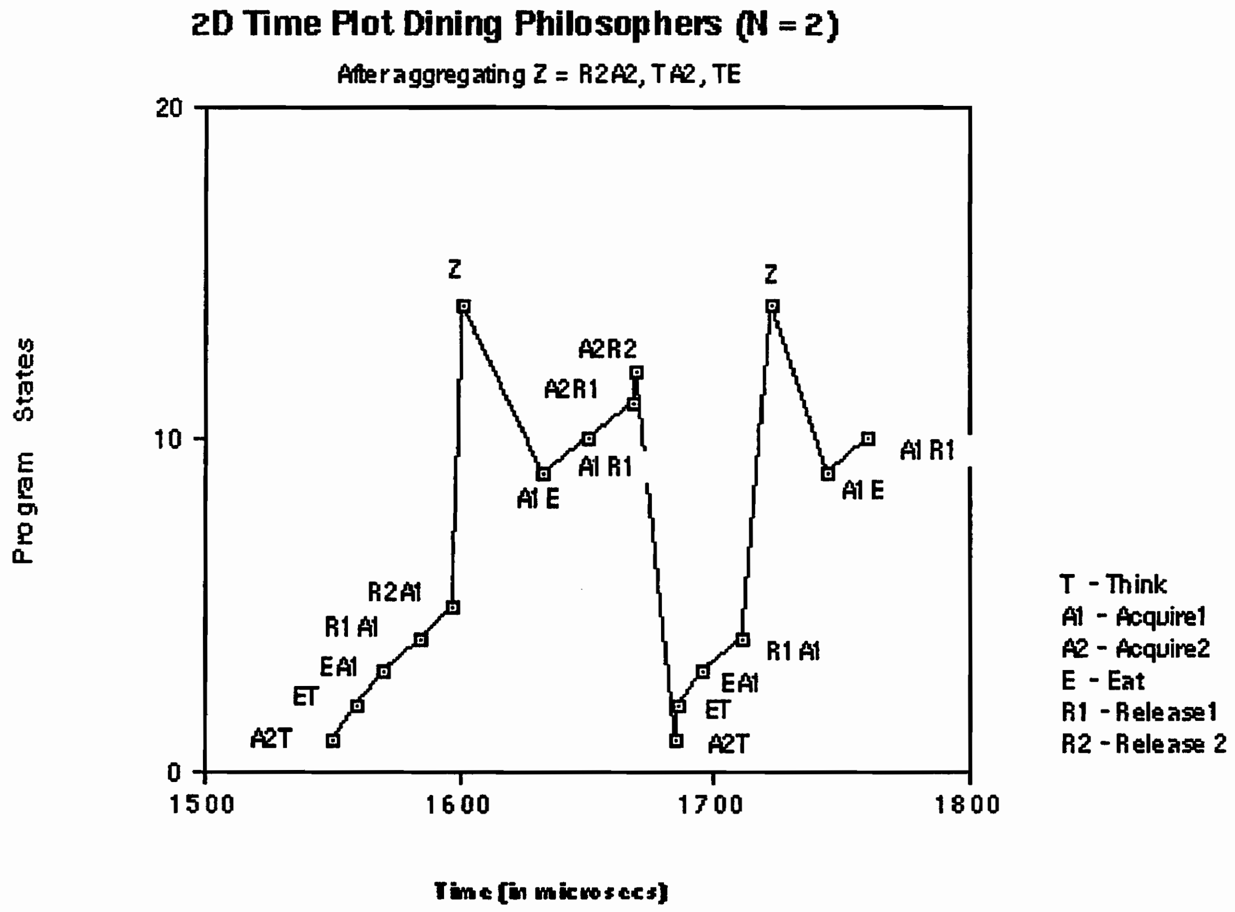


Figure 3.7: 2D Time Plot 2 philosopher Dining Philosophers problem – after aggregation.

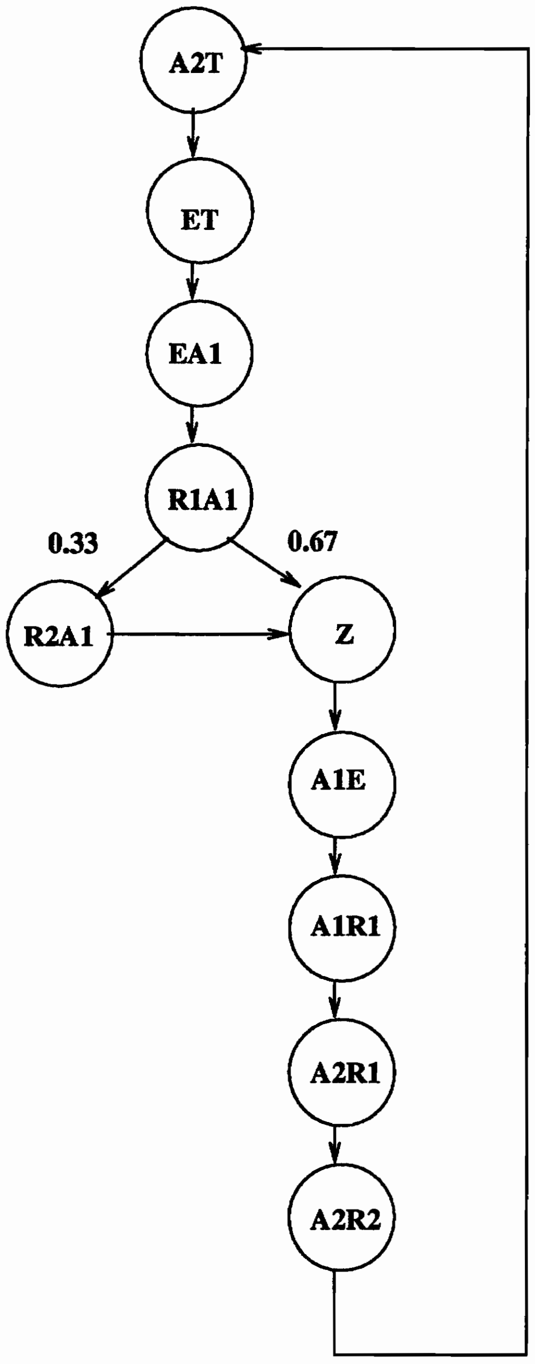


Figure 3.8: Semi-Markov chain model for a 2 philosopher Dining Philosophers problem – after aggregation.

and the program state occupancy time sequence is

$$O_o = (18, 17, 1, 16).$$

Figure 3.9 is a time plot of the resulting PES, O, after clipping I with $n_i = n_f = 8$.

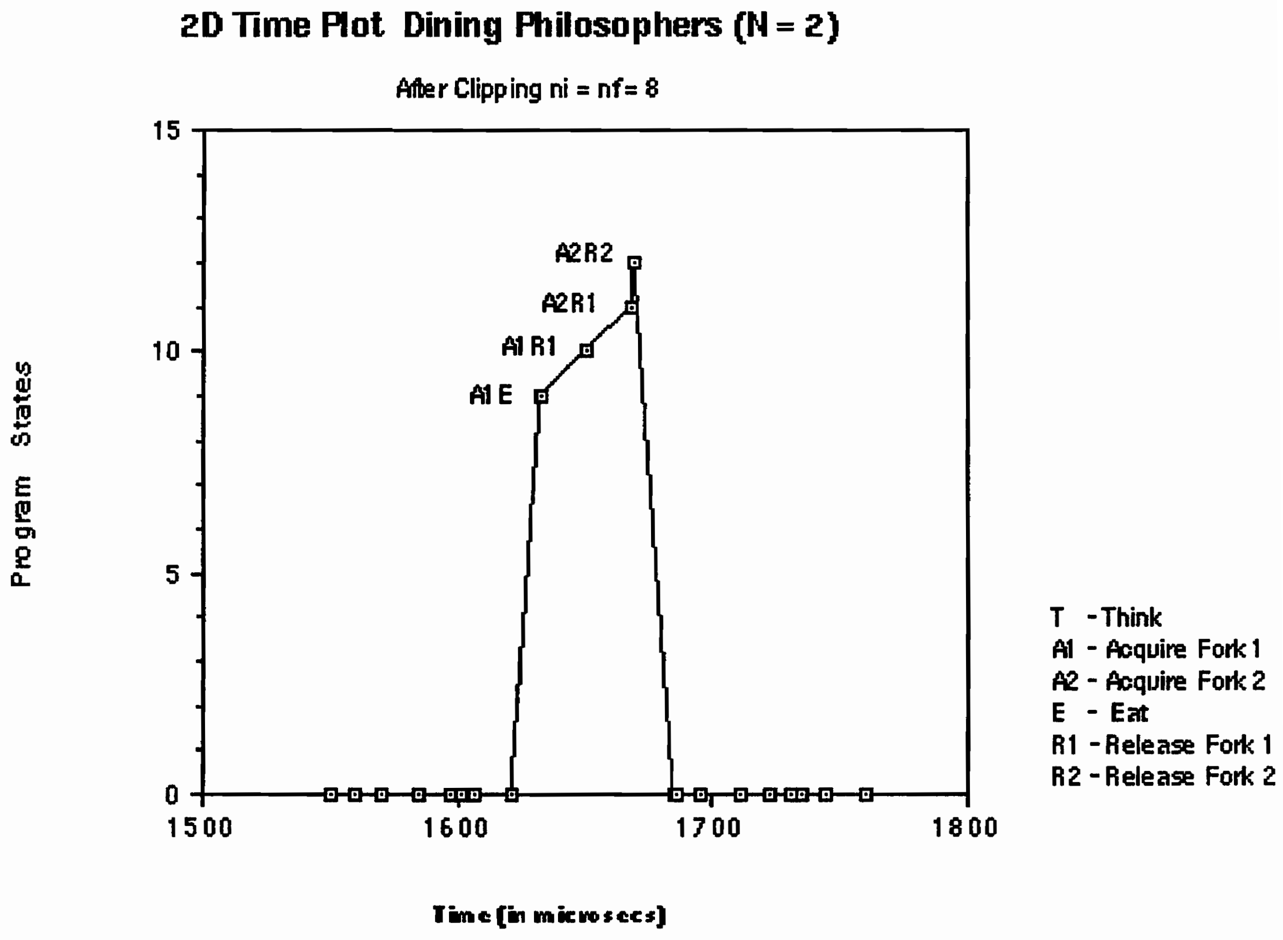


Figure 3.9: 2D Time Plot 2 philosopher Dining Philosophers problem – after event domain clipping.

3.3.4 Projections

Projections are special-purpose transforms that are defined by the user. A user may wish to provide transforms for specific types of PES's. As an example of a projection, consider a PES P which has m components and let us define a value i. A projection of this form takes the PES P and i as inputs, to create a new PES without the ith component of P.

3.4 Model Construction

The PES's and transformations discussed in sections 3.1 and 3.3 form the building blocks for constructing a semi-Markov chain model of the PES. The first step is to represent a program as a PES. The transformations are then applied on the PES. Initially the transient portions of the PES are clipped off. Following this, sequences of program states that occur frequently are aggregated and randomly occurring sequences are filtered from the PES. This process is repeated until the user is satisfied with the semi-Markov chain model generated by the resultant PES. The PES initially consists of only program states. At the end of the transformation process, the PES has three different kinds of states: basic program states, filter aggregate states, and pattern aggregate states. Figure 3.10 shows a flowchart of this process.

The semi-Markov chain model is constructed by mapping every state in the final PES to a stochastic state. For each stochastic state, the mean occupancy time and transition probabilities are computed from the PES.

Example 3.4.1 *Figure 3.11 shows the semi-Markov chain model for the PES of the two-philosopher Dining Philosophers problem after applying the time-domain and event-domain filters in Examples 3.3.1 and 3.3.2.*

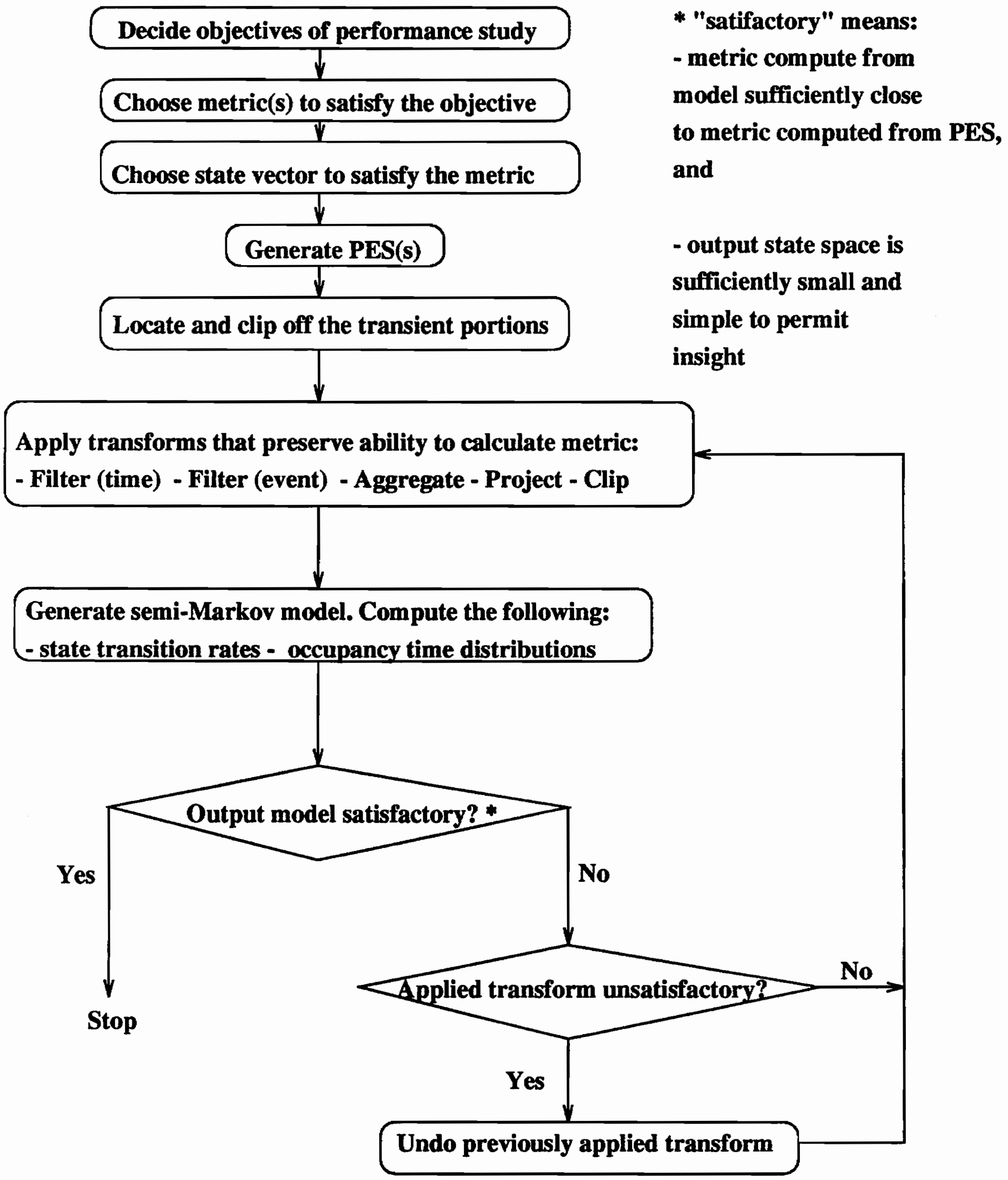

Decide objectives of performance study
Choose metric(s) to satisfy the objective
Choose state vector to satisfy the metric
Generate PES(s)
Locate and clip off the transient portions
Apply transforms that preserve ability to calculate metric:
- Filter (time) - Filter (event) - Aggregate - Project - Clip
Generate semi-Markov model. Compute the following:
- state transition rates - occupancy time distributions
Output model satisfactory? *
Yes
No
Stop
Applied transform unsatisfactory?
No
Yes
Undo previously applied transform
* "satifactory" means:
- metric compute from model sufficiently close to metric computed from PES, and
- output state space is sufficiently small and simple to permit insight

Figure 3.10: Flowchart for model construction.

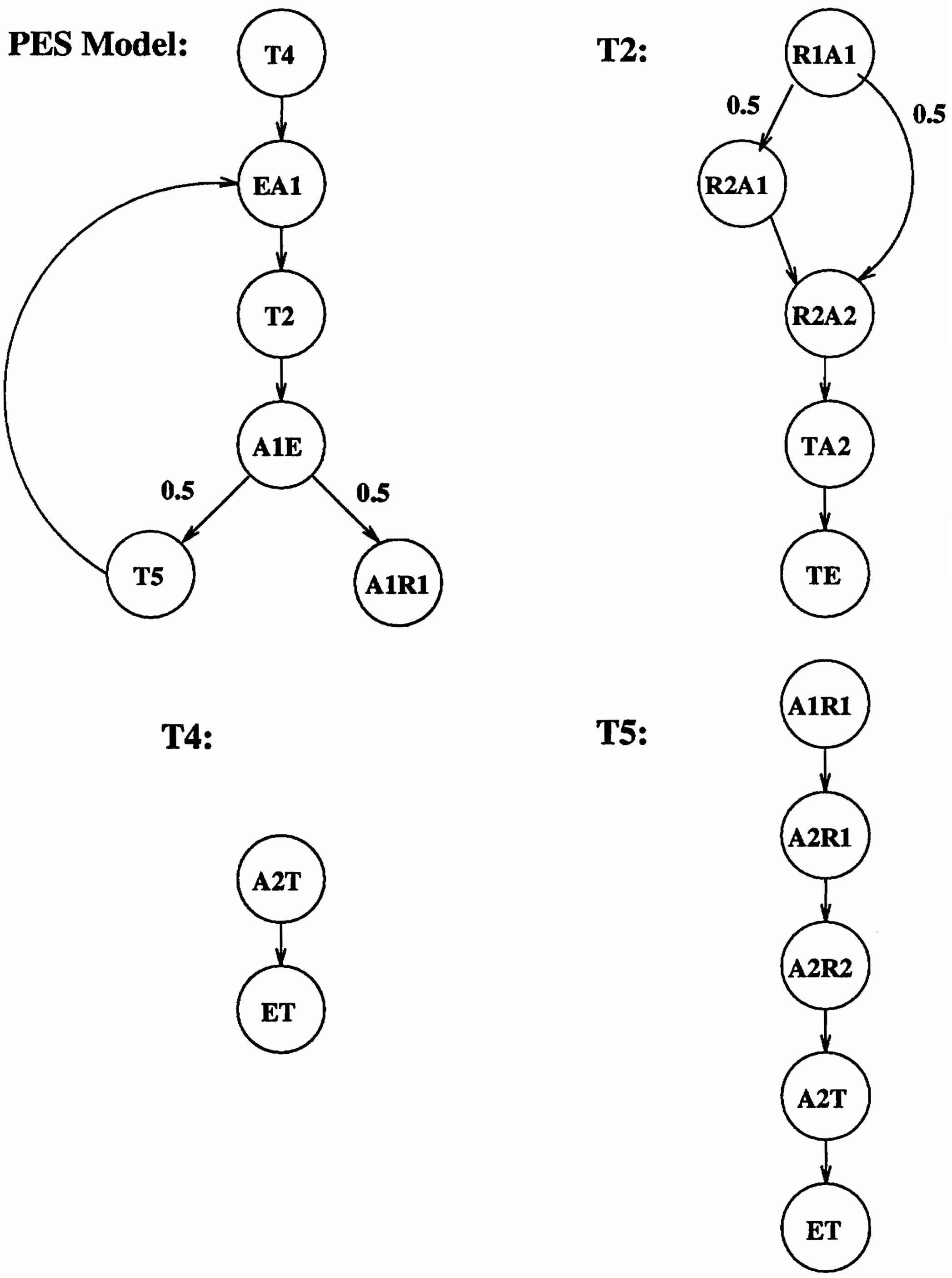


Figure 3.11: Semi-Markov chain model.

3.5 Periodicity in PES's

Most physical phenomena exhibit cyclic behavior of some sorts. As part of our analysis, we are interested in determining whether a PES exhibits cyclic behavior. To do this, we use Discrete Fourier Transforms (DFTs) for analyzing periodicity in a PES. We review DFTs briefly in the next section.

3.5.1 Discrete Fourier Transforms

Consider a series of real numbers $g_0, g_1, \ldots, g_{N-1}$. We can write the series formally for $0 \le j \le N-1$,

$$g_j = a_0 + \sum_{k=1}^{N/2} a_k \cos(kj \cdot 2\pi/N) + \sum_{k=1}^{N/2} b_k \sin(kj \cdot 2\pi/N).$$

The upper summation limit $N/2$ should be written as $\lfloor N/2 \rfloor$, and if N is even, we omit the $b_{N/2}$ term. The numbers a_k and b_k are defined by

$$\begin{aligned} a_0 &= \frac{1}{N} \sum_{j=0}^{N-1} g_j, \\ a_{N/2} &= \frac{1}{N} \sum_{j=0}^{N-1} g_j \cos(j\pi) \end{aligned}$$

and for $1 \le k \le N/2$, we have

$$\begin{aligned} a_k &= \frac{2}{N} \sum_{j=0}^{N-1} g_j \cos(jk2\pi/N), \\ b_k &= \frac{2}{N} \sum_{j=0}^{N-1} g_j \sin(jk2\pi/N). \end{aligned}$$

The a's and b's are called the finite, or discrete Fourier transforms of the g's and the latter are called the inverse finite (discrete) Fourier transforms of the a's and b's. For the discrete Fourier Transforms, the g values are assumed to have been sampled every Δ time units. The quantity $T \equiv N\Delta$ is called the *record length* or *fundamental period.*

The fundamental circular frequency is defined as $1/T$. Therefore, at the point $j\Delta$, the sum defining g_j is taken over frequencies that are integer multiples of the fundamental frequency. The highest multiple of the fundamental is $N/2$ corresponding to a highest frequency of $N/2 \cdot 1/T \equiv 1/(2\Delta)$, called the *Nyquist frequency.*

Power for a Discrete Series

The power at frequency k for a discrete sequence is

$$\begin{aligned} P_0 &\equiv N a_0^2, \\ P_k &\equiv \frac{N}{2}(a_k^2 + b_k^2), 0 \le k \le N/2 \end{aligned}$$

where $0 \le k \le N/2$, but when N is even

$$P_{N/2} \equiv N a_{N/2}^2.$$

A plot of the power versus the frequency of a discrete series is called a *periodogram.* Power is plotted against frequency k/T which varies from 0 to $1/(2\Delta)$.

3.5.2 Periodograms in Chitra

Chitra supplies two types of periodograms – periodograms in the event domain and periodograms in the time domain, to assist in identifying periodicity in a PES.

Event domain Periodograms: In the event domain, the g_i values are obtained from the value of each individual, distinct component in the PES. The timestamp of each state vector is ignored. Thus, the value of N is determined by the size of the PES itself. To adapt these values to the discrete fourier transform, Δ is assumed to be 1. The periodogram is plotted against frequency which varies from 0 to 1/2.

Time domain Periodograms: In the time domain, the value of Δ (the sampling duration) can be pre-set as can the value of N (the number of samples). The periodogram is plotted against frequency which ranges from 0 to $1/(2\Delta)$.

3.6 Summary

This chapter presents the methodology underlying the CHITRA92 performance analysis system for parallel programs. Some of the salient features of the Chitra methodology are as follows:

- A program's execution is represented as a program execution sequence (PES) and an associated program parameter set.
- A program can be modeled as a continuous-time homogeneous semi-Markov chain.
- It is desirable to *transform* a PES before mapping it to a semi-Markov chain model. Four different transforms have been developed: filtering, clipping. aggregation and projections.
- The Chitra methodology has a novel approach to identifying periodic behavior in a PES: a periodogram of the DFT of the PES is computed and the peaks in the periodogram are used to determine frequencies at which periodic behavior is suspected.

Chapter 4

The CHITRA92 User Interface Metaphors

Chapter 3 discussed the methodology of the Chitra performance analysis environment. This chapter, describes the realization of the methodology of Chapter 3 as a concrete system. More specifically, we describe tangible metaphors for the concepts presented earlier. The CHITRA92 system allows the user to manipulate *instances* of a PES; an *instance* can be thought of as user concept that is useful to explore multiple transformation paths of a PES. Users may load multiple instances of the same PES, and manipulate them differently.

4.1 The CHITRA92 User Interface Principles

The CHITRA92 user interface is founded on the following principles:

- The CHITRA92 user interface should provide a unified and consistent approach to realizing the Chitra methodology.
- The CHITRA92 system should provide the ability to to display a single PES instance or several PES instances simultaneously.
- The CHITRA92 system should provide the ability to compare different sequences of transforms of the same base PES and compare the resultant models.
- The CHITRA92 system should allow the user to select and apply transforms (discussed in Chapter 3, on an instance of a PES. The ability to undo the effect of a transform should also be provided.
- The user should allow visual editing of PES instances. Transforms are normally invoked through the visualization interface.

- The user interface should permit expansion in terms of new transforms and new display mechanisms for PES's.
- The transforms should be orthogonal with respect to the views i.e. the same transforms should be applicable on a PES from different views.
- The user interface should be highly interactive and have as low a response time as possible.

4.2 CHITRA92 Views

The visualization system's primary responsibility is to display the PES instances that have been loaded into the system. It must also be able to display a PES instance in different windows at the same time. To do this, we introduce the idea of a *view*. A view is a visual representation of a PES instance, with a menu system that allows a user to manipulate the PES instance, and set options to control the display of the PES. The visualization system could therefore be described as a view manager that controls multiple views.

Figure 4.1 shows the relationship between multiple views and multiple PES instances. Several views can display information from a single PES instance, but the converse of a single view showing information from multiple PES instances is not allowed. The PES instance-view relationship is strictly one-to-many.

The CHITRA92 visualization system currently supports six different kinds of views. We provide a brief description of each view in the following subsections.

4.2.1 2D X-Y time view

A 2D X-Y time view displays a PES instance as a two-dimensional X-Y plot. Each program state is represented by a unique y-coordinate value. The association of y-coordinates to program states is arbitrary. The x-coordinate represents timing information. The distance along the x axis between two adjacent states is proportional to the time spent in the first state.

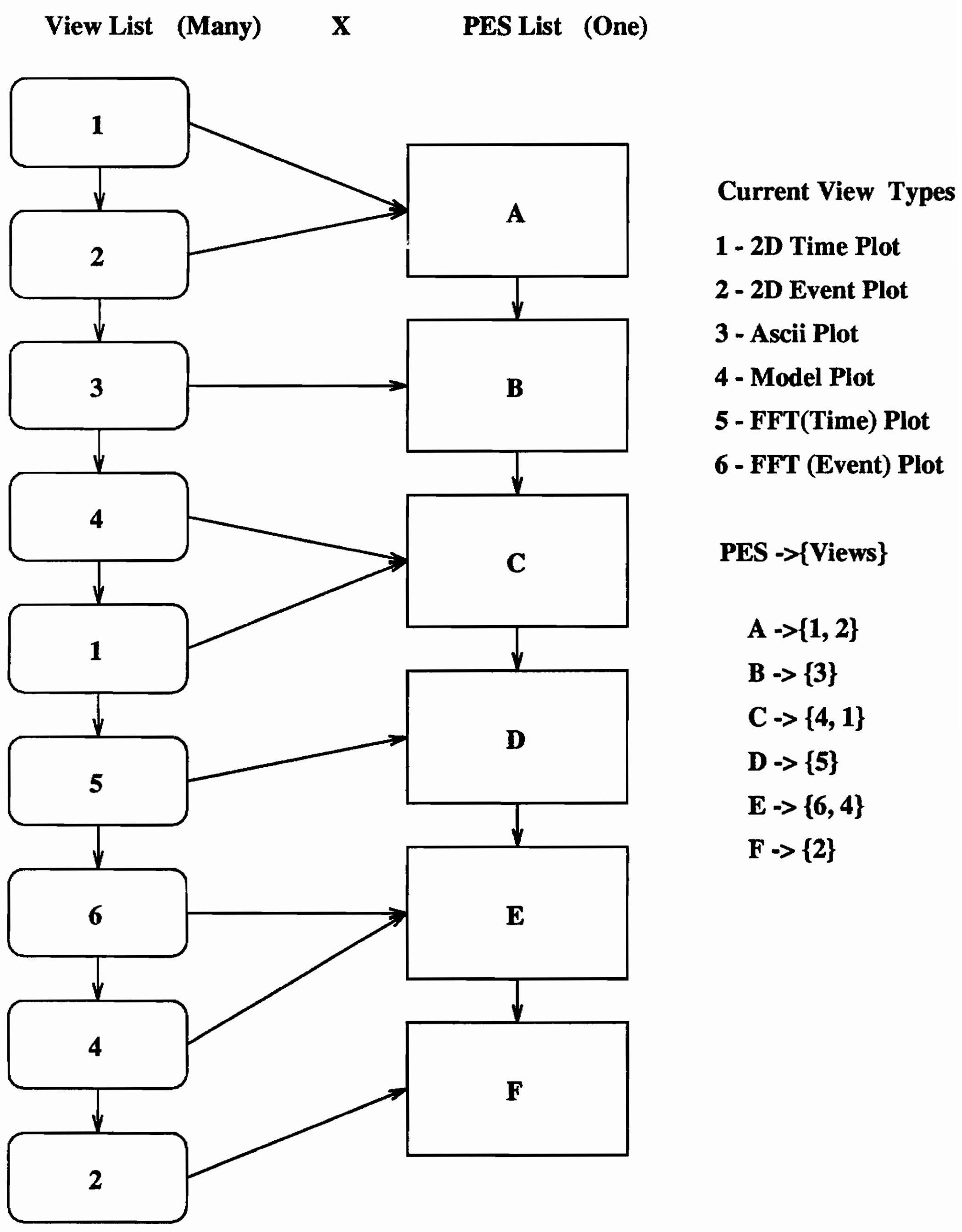


Figure 4.1: Relationship between multiple views and PES instances.

The 2D X-Y time view is useful in understanding the execution behavior of a program. It also conveys the relative occurrence of program states and the relative occupancy times. Figure 4.2 shows a 2D X-Y time view for a two-philosopher Dining Philosophers' problem.

The time view allows the user to access the transformation manager in the kernel. This allows the user to select a transform from the view menu and apply it on the PES instance being viewed. Interfaces to the filtering (time domain), aggregation and clipping (time domain) transforms are supported.

4.2.2 2D X-Y event view

A 2D X-Y event view is similar to a 2D X-Y time view. Again, each program state is represented by a unique y-coordinate value. However, the distance along the X-axis between adjacent states is always the same. This is because this view is intended to display the causal relationship between states. The event view also permits the user to access the transformation manager from its menu system. The event view menu provides interfaces to filtering (event domain), aggregation and clipping (event domain). Figure 4.3 shows a 2D X-Y event view for the two philosopher Dining Philosophers problem.

4.2.3 Text view

A text view displays a textual timestamped ordering of PES data records. Each line of information in the text view contains the following information:

- the data record number,
- the entry timestamp: time instant at which the program entered this program state,
- the program state id assigned by the PES mapping system,
- the type of the program state (whether it is an ordinary program state, a filter aggregate state, or a pattern aggregate state), and
- the state vector that the program state corresponds to.

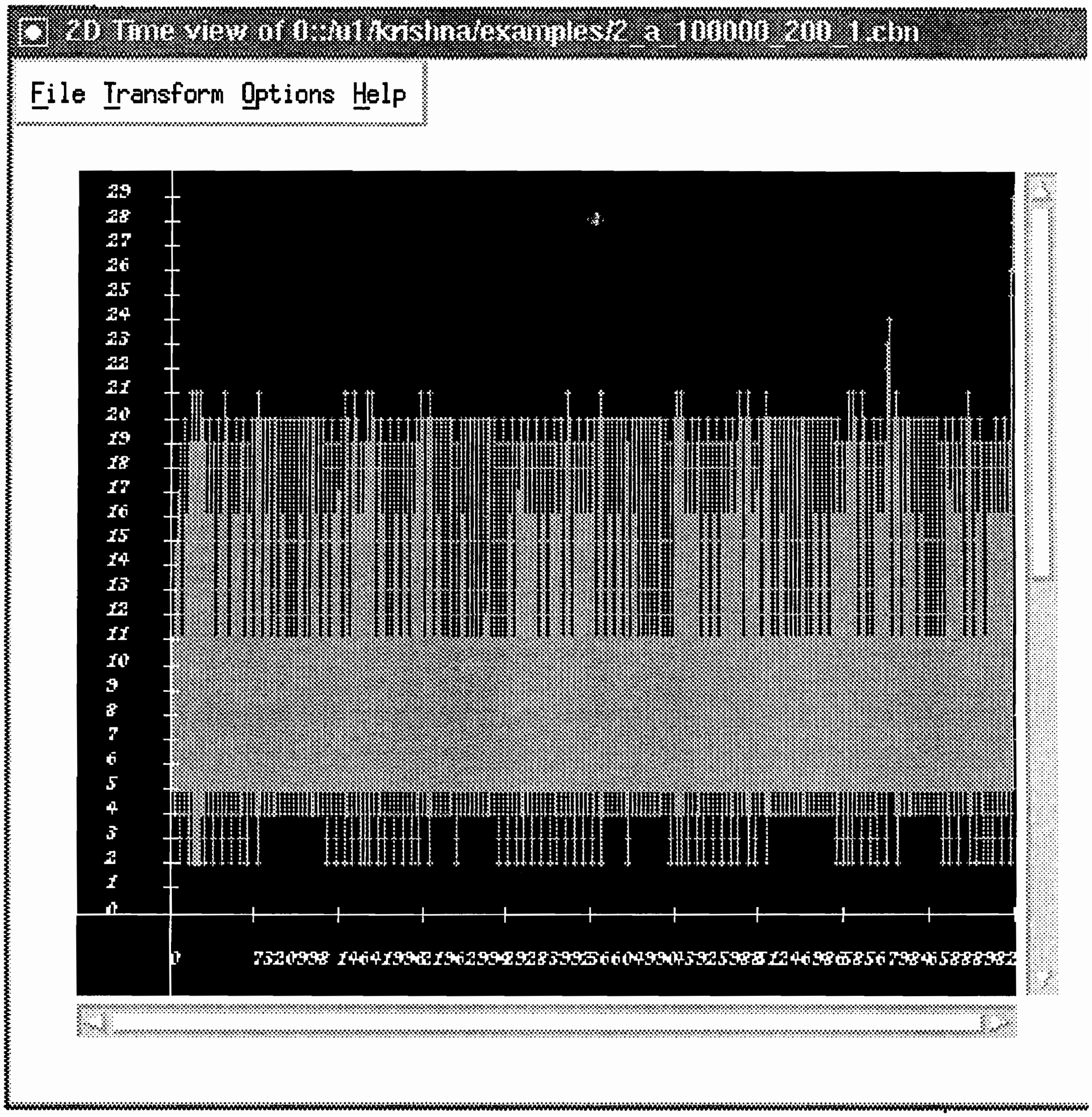

2D Time view of 0::/u1/krishna/examples/2_a_100000_200_1.cbn
File Transform Options Help
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0

Figure 4.2: 2D X-Y time view for the two philosophers Dining Philosophers problem.

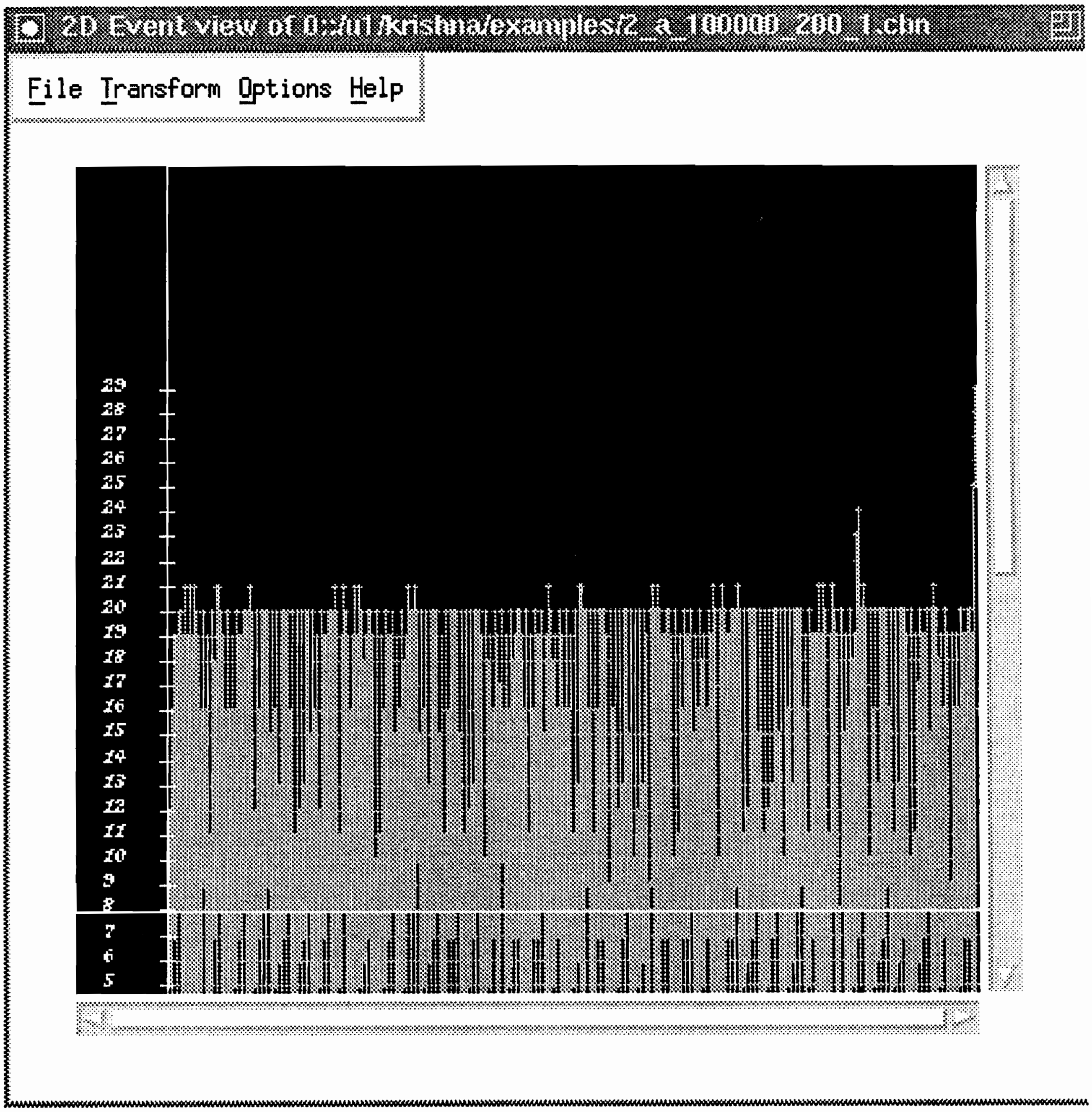
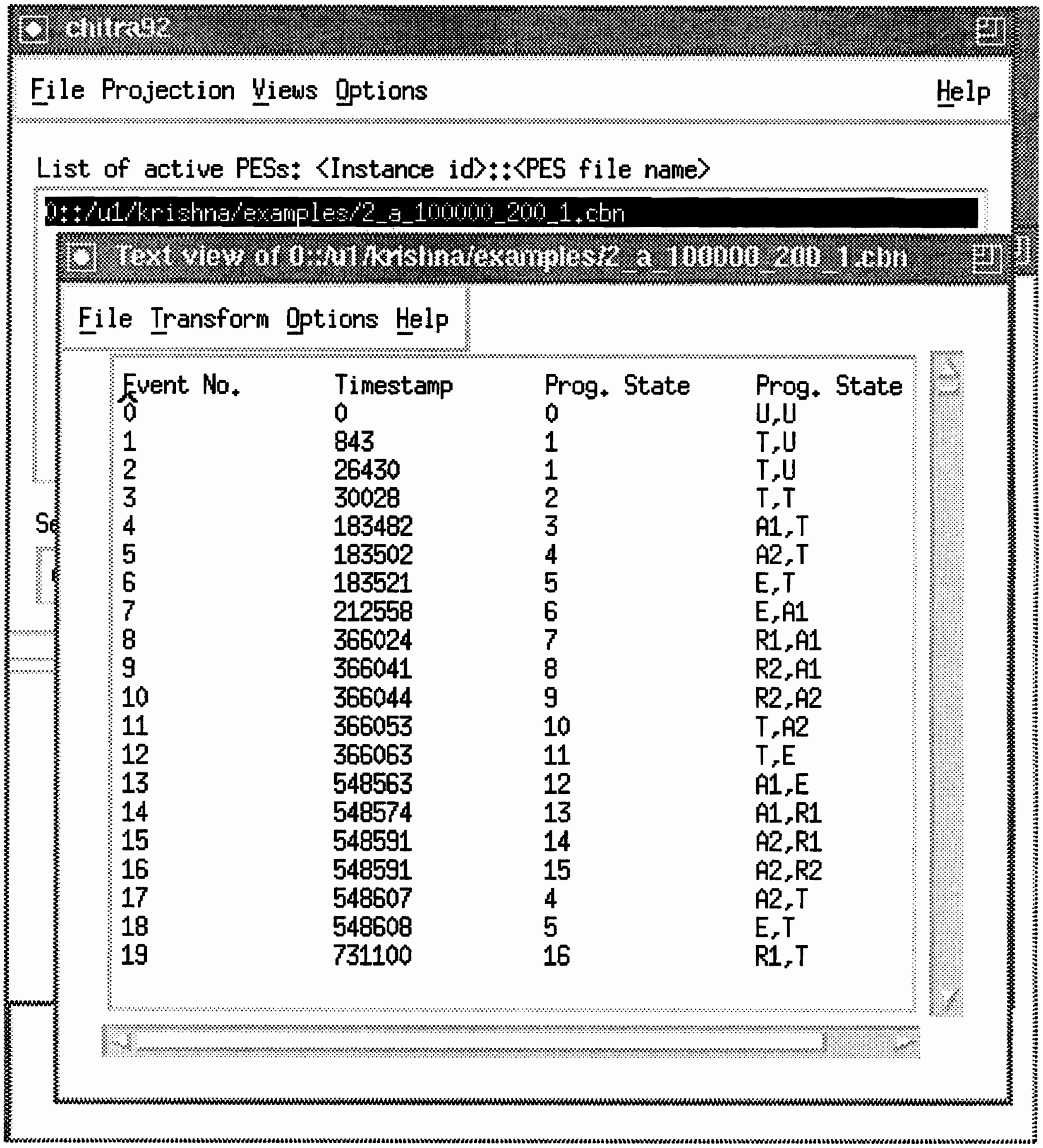

2D Event view of 0::/u1/krishna/examples/2_a_100000_200_1.chn
File Transform Options Help
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5

Figure 4.3: 2D X-Y event view for the two philosophers Dining Philosophers problem.


chitra92
File Projection Views Options Help
List of active PESs: <Instance id>::<PES file name>
0::/u1/krishna/examples/2_a_100000_200_1.cbn
Text view of 0::/u1/krishna/examples/2_a_100000_200_1.cbn
File Transform Options Help

```
Event No.     Timestamp    Prog. State    Prog. State
0             0            0              U,U
1             843          1              T,U
2             26430        1              T,U
3             30028        2              T,T
4             183482       3              A1,T
5             183502       4              A2,T
6             183521       5              E,T
7             212558       6              E,A1
8             366024       7              R1,A1
9             366041       8              R2,A1
10            366044       9              R2,A2
11            366053       10             T,A2
12            366063       11             T,E
13            548563       12             A1,E
14            548574       13             A1,R1
15            548591       14             A2,R1
16            548591       15             A2,R2
17            548607       4              A2,T
18            548608       5              E,T
19            731100       16             R1,T
```

Figure 4.4: Text view for the two philosopher Dining Philosophers problem.

Figure 4.4 shows the text view for the two-philosopher Dining Philosophers problem. Like the 2D X-Y views, the text view also provides interfaces to the transformation routines in the kernel.

4.2.4 FFT time view

The FFT time view displays a time domain periodogram of the discrete Fourier transform series of a PES instance. A histogram is used for displaying the periodogram, where the height of a bar indicates the cumulative power for the range of frequencies represented by the bar. The occurrence of spikes in the periodogram indicates a concentration of spectral energy for those range of frequencies which indicates that the PES has a high probability of being periodic at those frequencies. Figure 4.5 shows a time domain periodogram for the Dining Philosophers' problem.

The FFT time view menu system allows a user to set the sampling rate and also the number of samples that the user wishes to collect before computing the discrete Fourier transform.

4.2.5 FFT event view

The FFT event view displays an event domain periodogram of the discrete Fourier transform series of a PES instance. This is very similar to the FFT time view, the only difference being that samples for computing the discrete Fourier transform of the PES are taken on an event basis. Successive samples in the data set are computed using the values of successive program states. The FFT event view is therefore a fast but less accurate estimate of the PES's periodic behavior. Figure 4.6 shows an event domain periodogram for the Dining Philosophers problem.

4.2.6 Semi-Markov model view

The sixth and final view in the visualization system is the semi-Markov model view. This view displays the semi-Markov model of a PES instance as $N \times N$ matrix, $Model$,

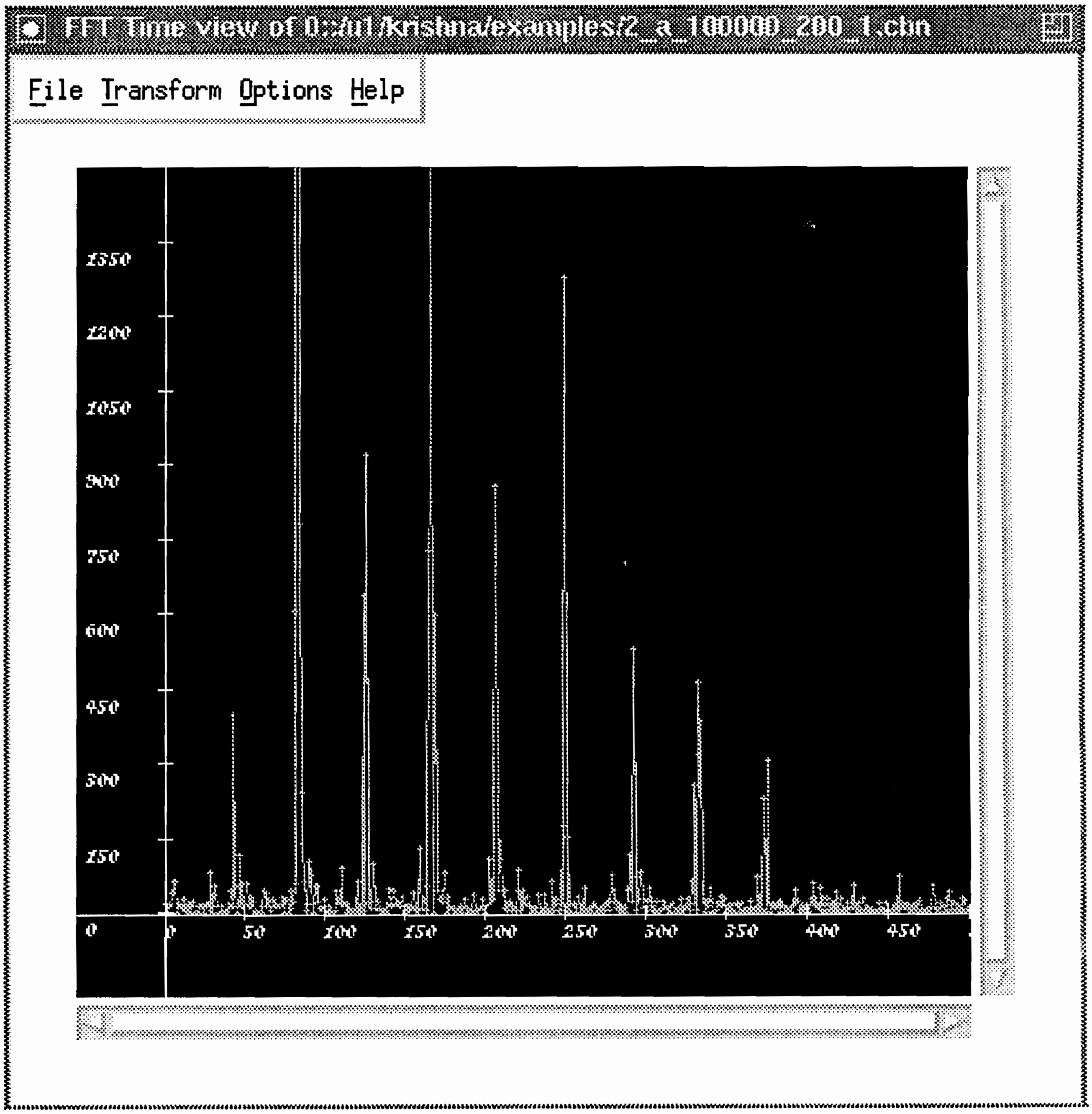

FFT Time view of 0::/u1/krishna/examples/2_a_100000_200_1.chn
File Transform Options Help
1350
1200
1050
900
750
600
450
300
150
0
0
50
100
150
200
250
300
350
400
450

Figure 4.5: Periodogram (time domain) view for the two philosopher Dining Philosophers problem.

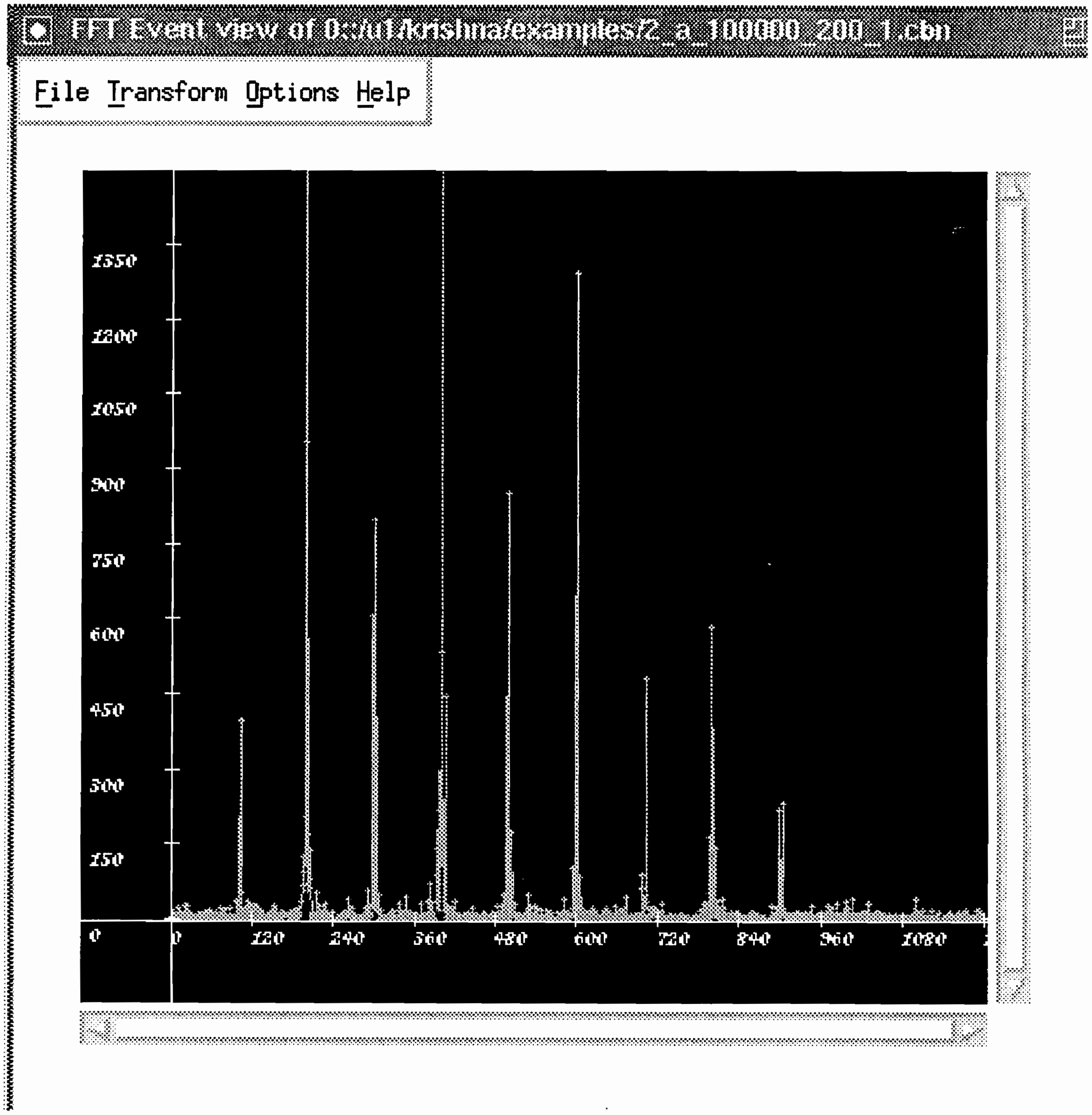


Figure 4.6: Periodogram (event domain) view for the two philosopher Dining Philosophers problem.

where N is the number of distinct program states in a possibly modified PES instance, and $Model(i,j)$ is the probability that the program makes a transition from the program state i to the program state j. Figure 4.7 shows a semi-Markov model view for the Dining Philosophers problem.

4.3 Transformation Metaphors

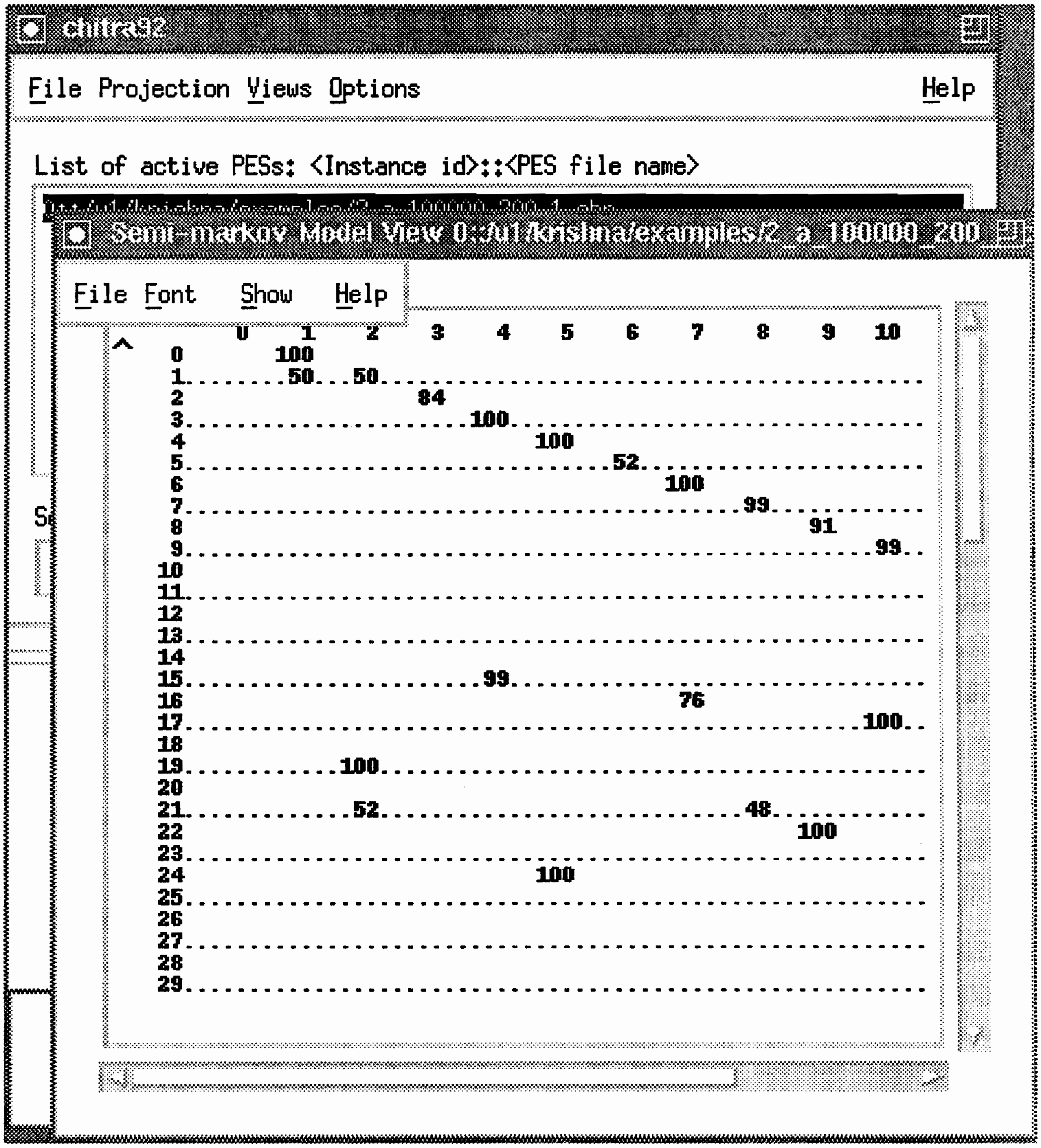
For every transformation defined in Chapter 3, there exists an equivalent user-interface metaphor. We describe them as follows:

1. Clipping: In the 2D X-Y time and event views a bounding box is selected and the region outside the bounding box is removed from the PES instance. In the Text view, clipping can be performed in a finer fashion: here the user specifies the specific range of data records in the PES instance to be retained, either textually or by highlighting the region to be retained with the mouse.

2. Aggregation: In the 2D X-Y time and event views, a bounding box is drawn around the region that contains a subsequence of program states in a PES instance. The selected subsequence of states is now used as a pattern to identify all other identical subsequences to aggregate. In the text view this is performed by selecting with the mouse the sequence of states that is to be used as a pattern.

3. Filtering: For all views, filtering is facilitated by displaying a histogram containing either the distribution of program state occurrence counts or occupancy times. In the time domain, the user is allowed to select a threshold occupancy time value and in the event domain, a threshold occurrence count value for filtering.

4. Projections: Projections are facilitated by providing the user with dialog boxes to input the name of the instance and associated parameters to the projection. There is no visual representation for a projection transform.


chitra92
File Projection Views Options Help
Semi-markov Model View 0:/u1/krishna/examples/2_a_100000_200
File Font Show Help

List of active PESs: <Instance id>::<PES file name>

```
          0    1    2    3    4    5    6    7    8    9   10
   0         100
   1.........50...50..................................................
   2                   84
   3.........................100....................................
   4                             100
   5.....................................52..........................
   6                                      100
   7..................................................99............
   8                                                       91
   9..........................................................99..
  10
  11...............................................................
  12
  13...............................................................
  14
  15..........................99..................................
  16                                       76
  17.........................................................100..
  18
  19...............100............................................
  20
  21...............52..............................48.............
  22                                                      100
  23...............................................................
  24                             100
  25...............................................................
  26
  27...............................................................
  28
  29...............................................................
```

Figure 4.7: Semi-Markov model view for the two philosophers Dining Philosophers problem.

The process of modifying a PES via a graphical interface is called *visual editing* of a PES. Thus the transforms can be applied on a PES visually or textually.

4.4 Summary

The important features of the CHITRA92 user interface are as follows:

1. A user is allowed to load multiple instances of a PES into the system.
2. The user interface provides the ability to apply and/or undo transformations on a PES instance, and to generate a semi-Markov chain model of the PES instance.
3. The CHITRA92 visualization system provides the top-level interface to the system and access to the PES instances via views. Views provide interfaces by which the user can apply a transformation on a PES instance. The views supported include: 2D X-Y time views, 2D X-Y event views, Text views, FFT (time) views, FFT (event) views and semi-Markov model views.
4. The X-Y views represent temporal and causal relationships of program states within a PES.
5. The Text view is an time-ordered textual display of a PES instance.
6. The FFT views display periodograms in the time and event domains.
7. The semi-Markov chain represents the state transition matrix of a PES instance.

Chapter 5
The Architecture of the CHITRA92 System

Chapter 3 discussed the Chitra methodology to model concurrent software. Chapter 4 presented the user interface of the system. In this chapter, we describe the architecture of CHITRA92 that implements the user interface described in Chapter 4.

This chapter discusses the overall design of the system, the internal components, and other implementation specifics. Section 5.1 describes the design goals that guided the implementation of the CHITRA92 system. Section 5.2 gives an overview of the CHITRA92 system software. The system is divided into three functional components. The first component is the *pre-processor software* which is responsible for representing performance data as PES's. The pre-processor software includes a translator that converts a human-readable, textual specification of a PES to binary form. The second software component is the CHITRA92 *kernel*. The kernel is responsible for managing multiple instances of PES's that have been loaded into the system, applying transforms to PES instances and keeping track of the list of transforms that have been applied to a modified PES, and generating the semi-Markov model of a reduced PES. Finally, the third component of the system is the *visualization system* which supports a variety of graphical displays of PES data. Sections 5.3, 5.4, and 5.5 describe the CHITRA92 pre-processor software, the CHITRA92 kernel, and the CHITRA92 visualization system respectively.

5.1 Design Goals of CHITRA92

The fundamental requirements of the CHITRA92 system architecture are that it be scalable, modular, portable and versatile. We explain each requirement in detail.

1. **Scalability**

   There are two issues to be considered here.

   The first issue is with respect to the performance analysis environment itself. The system should permit multiple PES's to be analyzed during a single session. There should also be no problem in displaying multiple views of the same PES simultaneously. Because PES analysis is always done in comparison to the analysis performed on other PES's with the same or with different input parameter sets, simultaneous viewing of PES's is very important.

   The second issue is more difficult to address. With the trend towards massively parallel computing, the number of processors on most MIMD architectures ranges from two to greater than 2000 processors [1]. By increasing the number of target processors, a programmer increases program performance without making significant changes to the hardware or software. As the parallel system scales in size and performance, the performance analysis environment should scale as well. Scaling the performance environment implies that the system should be able to effectively capture and display data from a large number of processors. Current display and analysis techniques definitely do not scale commensurately for massively parallel applications [22].

2. **Modularity**

   Our experience with performance analysis systems for parallel programs suggests that while parallel architectures and programming paradigms are fairly well established, novice users rarely get access to such computing. It is only fairly expert programmers and researchers who require and have access to massively parallel computing. Such users are concerned with extracting the maximum amount of performance from their programs and the underlying architecture. At the very minimum, these users are concerned with identifying performance bottlenecks, computing performance metrics and comparing these metrics with other metrics. Such users require an enviroment that provides a collection of tools that can be configured to suit their purpose.

Secondly, performance data comes in a variety of formats and the requirements of performance environment users are too complex to be completely provided for by any system. Ultimately users will wish to modify the system itself.

It is important, therefore, that the user be allow to extend or modify the system (right down to the level of system source code). In order to do so, the system must be highly modular and permit modifications to be performed quickly and seamlessly.

3. **Portability**

Any tool that expects some semblance of recognition among the research community must be widely and easily available. With the increasing number of diverse programming environments available to researchers, a performance tool should be portable and simple to use.

Performance analysis tools for parallel programs have two primary goals. The first goal is to allow an application developer to fine-tune his application program in order to obtain maximum performance on a particular target architecture. The second goal is to ensure that the application can be run on an variety of architectures and that the the performance of the application can be studied across different architectures,

For both goals, it is necessary that the same performance analysis studies be performed on different platforms. Because performance analysis is probably the last resort for an application developer, it is very important to see that the same performance evaluation tools are present on all platforms.

4. **Data-transportability**

The performance analysis system should be designed so that it accepts data in a transparent manner. It should not matter where the performance data comes from. For example, consider a program executing on a shared bus multiprocessor computer system like the Sequent Symmetry. To conduct a performance study of this program, we collect performance data from the Sequent Symmetry. However, the performance analysis environment resides on a DECStation 5000 system. For the performance environment to be data-transportable, it should be possible to transfer the data from the Sequent to the DECStation and still be able to analyze it.

5.2 CHITRA92 Software Overview

As described in the previous section, the fundamental design goals in the design of the CHITRA92 system are that it be portable, scalable, modular, and versatile. In this section, we describe the functional components of the system and show how they together satisfy the design requirements outlined in the previous section. The CHITRA92 software consists of three major components: a pre-processor software system, a kernel that performs analyses and transformations on PES's and a visualization system. Figure 5.1 shows the organization of these three components with respect to one another.

In Section 5.1, we said that the system should be portable across multiple platforms. For the CHITRA92 system to be portable implies that the PES's must also be portable. Portability is best achieved by using a textual representation of the PES. Text files, however, are neither efficient to access nor are they compact to store (a compact representation is required because PES's may be many megabytes in size). A binary representation format satisfies both these requirements. However, differences among machine architectures such as byte-ordering and word sizes make binary representations non-portable. The CHITRA92 pre-processor software has been designed to eliminate this problem. The pre-processor software provides a specification language (Chitra Specification Language) which allows a user to describe the structure and record the actual data of a PES in text format. A translator which converts the text specification to a binary representation of the PES is also provided

The second major component is the CHITRA92 kernel. The primary goal of the CHITRA92 system is to model the input PES as a semi-Markov chain after transforming it suitably. Therefore, the system should provide a set of transformation techniques that the

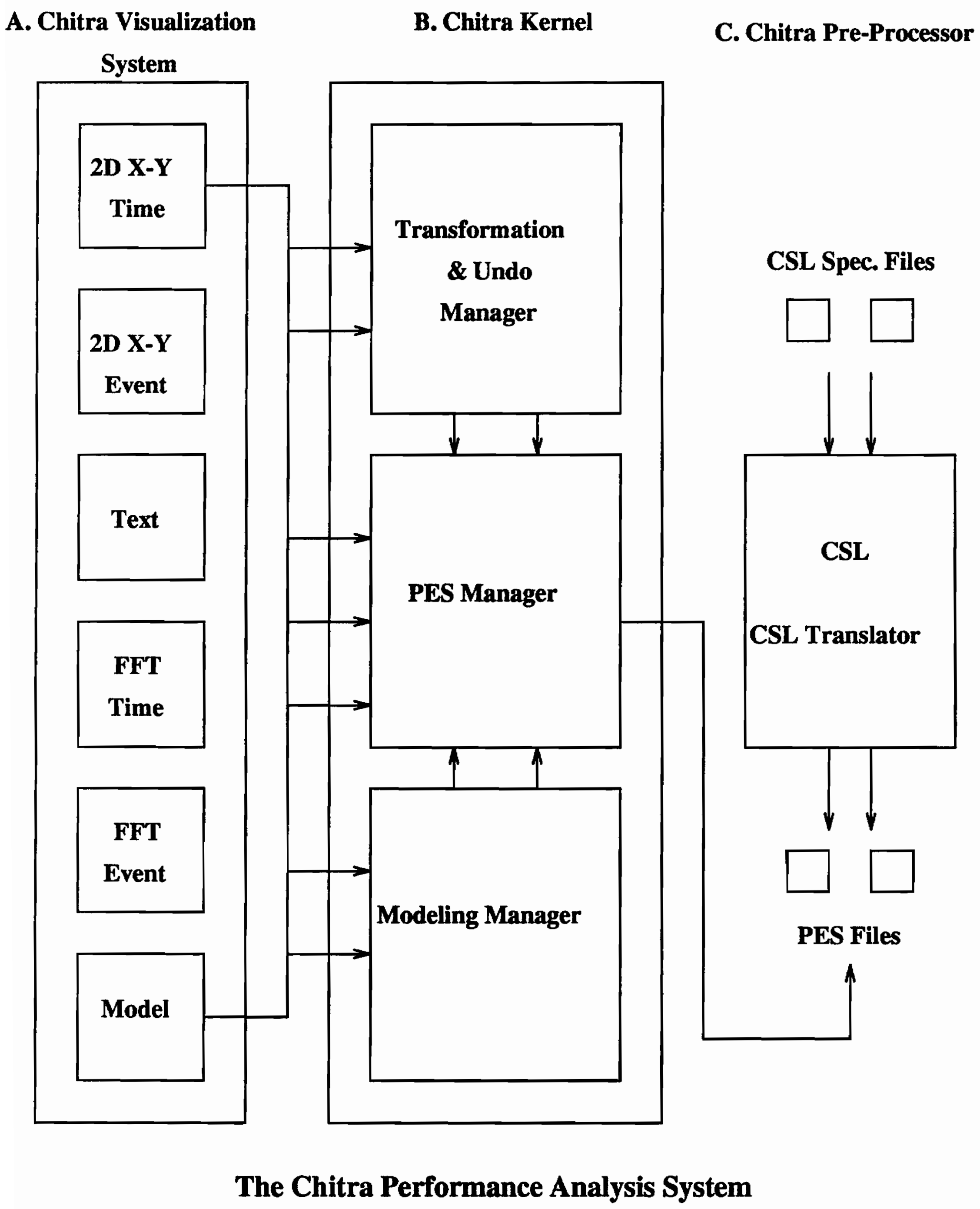


Figure 5.1: The CHITRA92 Performance Analysis System.

user can select and apply on a PES. Once the PES has been modified, the semi-Markov chain model must be generated. The CHITRA92 kernel provides both a transformations subsystem and a modeling subsystem for this purpose. The CHITRA92 kernel also manages multiple instances of PES's and keeps track of the list of transforms that have been applied to PES instances.

The third software component is the CHITRA92 visualization system. Performance data from programs running for a few seconds can be several megabytes in size. It is difficult to comprehend large volumes of data textually. Graphical displays assist users in a major way by providing meaningful displays that can be quickly comprehended. Visual displays are also useful for displaying information such as the distribution of the number of occurrences of program states or the occupancy time spent in a program state. The CHITRA92 visualization system also provides a set of graphical displays of the PES which allow the user to select and apply transforms on a PES. Currently CHITRA92 supports six different displays. Each display is called a *view*. The different types of views are 2D X-Y time views, 2D X-Y event views, text views, FFT (time) views, FFT (event) views and semi-Markov model views. Each of these views have appropriate interfaces to the transformations module in the CHITRA92 kernel.

The visualization system is also responsible for providing the user with a top-level interface, information displays for pattern aggregates and filter aggregates, color and font selection mechanisms, and other miscellaneous interfaces to the CHITRA92 kernel.

The entire CHITRA92 system has been implemented in C, C++, and Fortran. The visualization system uses the OSF/Motif 1.1 toolkit*, the MIT Athena toolkit, and the X11 Window system†. Because of the wide-spread availability of the X11 Window System and the Motif toolkit, portability across a wide-range of systems is achieved.

Also, all of the visualization and transformation modules are self-contained and independent. While the current version of the CHITRA92 system runs on a single workstation,

*OSF/Motif and Motif are trademarks of the Open Software Foundation, Inc..
†The X Window System is a trademark of the Massachusetts Institute of Technology.

it should not be difficult to implement a distributed version of the system, ensuring that the performance analysis environment scales commensurately with the parallel program being analyzed.

5.3 CHITRA92 Pre-processor Software

The CHITRA92 pre-processor software is responsible for preparing a program execution sequence (PES) for the CHITRA92 kernel. Chapter 3 defines a PES qualitatively to explain the Chitra methodology. In this section, we describe it from an implementation standpoint.

The PES representation of CHITRA92 differs from the representation that most other performance analysis and visualization systems use. Most performance analysis tools for parallel programs view the parallel program as a sequence of events. The parallel program is instrumented in a non-intrusive manner to produce a trace file of events. Each trace file record consists of a globally synchronized timestamp, a processor id, an event id, and associated event specific information.

The CHITRA92 PES, on the other hand, consists of a number of data records. Each record consists of a timestamp and a state vector. Each state vector consists of a user-defined number of components. Each component represents some piece of program information (e.g. local or global variables, message queues, a pointer to the next instruction being executed in a thread). An example is recording the position of the code pointer in all concurrently executing threads of a parallel program. Other systems would have to use a single record for each thread's code pointer because of the restriction that each record can record data only for a single process. The CHITRA92 PES is, in a sense, a *spatially* enhanced trace file, because it provides information at a given timestamp for all tnreads in a single data record.

The CHITRA92 pre-processor software is responsible for the following actions:

1. **Chitra Specification Language:** The Chitra Specification Language is a PES description language that allows a user to describe the structure and the data of a PES. CSL supports specifying the number and data type of the components of the PES

state vector.

2. **The CSL Translator:** The CSL specification of a PES allows portability because of its text format. However, because text data cannot be efficiently processed, a translator is provided for the CSL language which allows the user to convert from text to binary formats.

In the following subsections we briefly discuss parallel program instrumentation, the Chitra Specification Language and the CSL Translator. The next subsection is a brief discussion on instrumenting parallel programs.

5.3.1 Instrumentation of Parallel Programs

Most performance analysis environments provide an instrumentation facility that assists in the generation of performance data from a target application. There are several ways to instrument programs. The easiest and most flexible method is a software technique – source code instrumentation. The programmer inserts instrumentation source code at appropriate points in his code, compiles the code and links it to a software instrumentation library with function calls that record event information at runtime. If the program is moved to another architecture, the instrumented version can still be used for generating performance data as long as the instrumentation library is also ported to the new platform. The problem with source code instrumentation is that it introduce perturbations that could distort the trace. If the error introduced in the generated trace data is minimal (a good estimate is less than 10 percent error [23, 31]) then the trace is assumed to be a fairly good estimate. Currently, most performance data capture systems provide source code instrumentation software [38, 37, 22]. Figure 5.2 shows the various stages of activity in instrumenting source code of parallel programs.

At the other end of the spectrum is hardware instrumentation. In a broad sense, hardware instrumentation implies using special hardware to capture program information. While hardware instrumented traces are very accurate representations of the program's behavior,

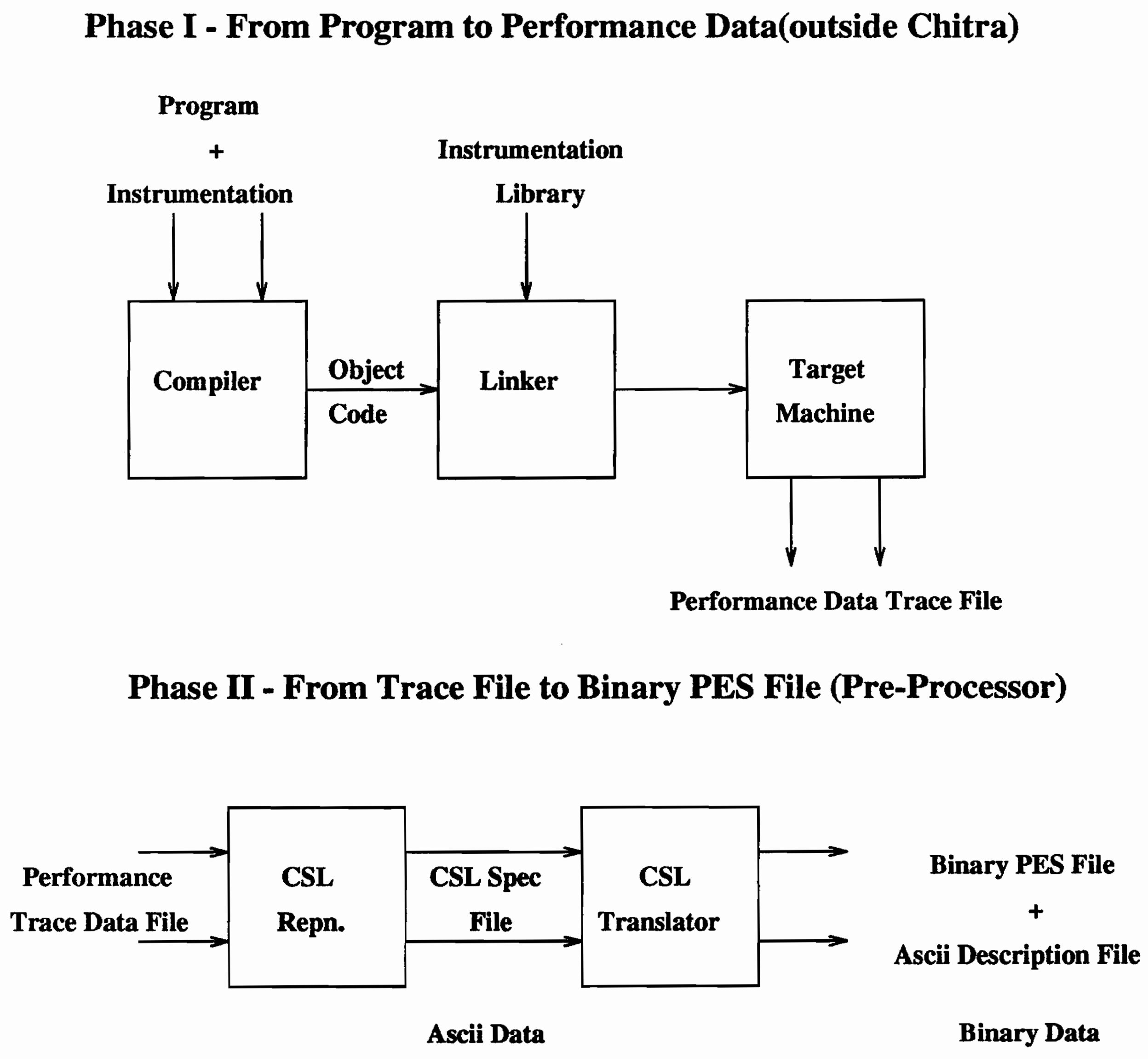

Phase I - From Program to Performance Data(outside Chitra)
Program + Instrumentation
Instrumentation Library
Compiler
Object Code
Linker
Target Machine
Performance Data Trace File
Phase II - From Trace File to Binary PES File (Pre-Processor)
Performance Trace Data File
CSL Repn.
CSL Spec File
CSL Translator
Binary PES File + Ascii Description File
Ascii Data
Binary Data

Figure 5.2: Phases in the instrumentation of programs.

hardware instrumentation implies non-portability because hardware vendors tend to provide non-standard trace monitoring hardware. Some of the more commonly known hardware instrumentation techniques include parallel traps, high-resolution real-time clocks, special-purpose registers and monitor buffers [25]. Currently, hardware instrumentation techniques are seldom used because of the effort involved in procuring this kind of special-purpose hardware. With the rapidly decreasing cost and proliferation of high-performance, application-specific, VLSI chips and as the need for accurate trace data increases, hardware monitoring should have an increased presence in the performance analysis field.

A compromise between hardware and software instrumentation techniques is also of interest. More and more of the instrumentation of programs can be automated with better compiler technology. By making the instrumentation facility visible to the compiler, optimized code can be generated keeping in mind the need for instrumentation (more memory buffers can be allocated to store portions of the event trace and file writes can be minimized).

While we have plans to integrate an instrumentation library in the future, CHITRA92 does not provide one. This is because the instrumentation library introduces portability problems. However much one may wish for a portable trace data capturing process, one cannot ignore the fundamental differences between machine architectures which make this very difficult. Identifying similarities[‡] and creating a compatible set of instrumentation routines is left as an open problem.

Instead of an instrumentation library, CHITRA92 allows trace data in any form and from any source to be post-processed after the program's execution to generate a PES. The performance trace data is incorporated into a CSL specification for the PES. Next, the CSL specification of the PES is converted into a binary representation. The binary PES file is then loaded into the CHITRA92 system for visualization and analysis. In Figure 5.2, this is indicated as Phase II.

[‡]A good example of the difficulty involved is comparing shared bus multiprocessors and distributed message passing architectures to create a trace capture library with the same instrumentation routines.

5.3.2 Describing a PES with CSL

The single most important concern for representing performance data in CHITRA92 is to provide a format offering ease of representation and portability across multiple platforms. For CHITRA92 to be portable, the performance data must be machine and programming language independent. Therefore it is necessary to take into account differences in architectures such as byte ordering and word sizes.

The volume of performance data generated from instrumented parallel programs is usually very high (in the order of several million bytes of data). Therefore, the representation format should allow fast access to data records and minimize the storage space required for data records.

Finally, the representation format of the PES can change. More data types or data record types may be required. It should be easy to add new data record or component types or modify existing types to the PES.

CHITRA92 addresses all of the above issues by defining a PES description language – the Chitra Specification Language (CSL). A CSL translator, which converts a textual specification of PES to binary format, has been integrated into the system. A CSL source file specifies the structure and content of a PES.

A CSL specification file consists of the program name and list of sections. Currently, a CSL specification contains two sections. The first section is the **Setup** section and the second is the **Trace** section.

The **Setup** section describes the structure of the PES state vector. For the each component in the state vector of the PES, the data type of the component and the range of values that the component can take are specified. Optionally, the user can associate identifiers with each of the values of the component. The **Trace** section contains the actual PES data records. Each data record in the PES consists of a time stamp and a state vector of components as defined in the **Setup** section.

Figure 5.3 shows a sample CSL specification file. This file describes the structure and

```
Program Dining;
begin
          Setup:
                    maxcomps = 2

                    Component 0-1 = 'Philosopher', 8, char;
                              0 = Think, T;
                              1 = Acq1, A1;
                              2 = Acq2, A2;
                              3 = Eat, E;
                              4 = Rel1, R1;
                              5 = Rel2, R2;
                              6 = Undef, U;
                              7 = Finis, F

          Trace:
                    2720243847 < 0, 7 >
                    2720244697 < 0, 1 >
                    2720244727 < 0, 2 >
                    2720244747 < 0, 3 >
                    2720244766 < 0, 4 >
                    ................
                    ................
                    ................
                    ................
                    2720268758 < 1, 2 >
                    2720268767 < 1, 3 >
                    2720268776 < 1, 4 >
                    2720268801 < 1, 5 >
                    2720268810 < 1, 6 >
                    2720268819 < 1, 8 >

end.
```

Figure 5.3: Example CSL specification for a 2 philosopher Dining Philosophers problem.

content of a two philosopher Dining philosophers' PES.

Within the **Setup** section, the number of components is 2. Each component is described by a name ("Philosopher") and a type ("char"). Both components 0 and 1 have the same number of possible values (8). The individual values (and their associated identifiers) are 0 (Think, T), 1 (Acql, A1), 2 (Acq2, A2), 3 (Eat, E), 4 (Rel1, R1), 5 (Rel2, R2), 6 (Undef, U), and 7 (Finis, F).

The **Trace** section contains the actual program trace, every record consisting of a timestamp and the state vector of the program at that timestamp. The state vector is represented as $\langle x_0, x_1, \ldots, x_{n-1} \rangle$, where x_i is the ith component value. In the figure, the first data record in the **Trace** section is is 272024387$\langle 0, 7 \rangle$.

5.3.3 CSL Translator Output

Representing PES's as CSL specifications allows performance data to be portable across platforms. However text specifications are neither compact nor efficient to process. The CSL translator's primary purpose is to generate output representations of a PES that can be quickly processed by the kernel PES-reader modules. The CSL translator's primary purpose is to generate two output files, one binary and one text[§], that can be easily accessed by the system file reader modules. The text file is called the *setup file.* The setup file stores the program name, the number of components, the possible values that the component may take, and the long and short identifiers for each of the component's values. Since information in the setup file is read only once (when the PES is loaded into the system), this representation is acceptable from an efficiency standpoint. The binary file is called the *trace file.* The trace file consists of several records; each record consists of a time stamp and a state vector of size N where N represents the number of components specified in the CSL specification.

[§]For the release version ofCHITRA92, a PES will be represented as a single binary file

5.4 The CHITRA92 Kernel

The CHITRA92 kernel consists of four interacting subsystems that manage any number of instances of PES's that have been loaded into the system. The primary subsystem is the *PES manager* which performs several house-keeping functions such as loading a *PES instance* from its constituent PES binary and ASCII files, mapping the state vectors in the PES to program states based on an appropriate mapping function, managing multiple PES instances and invoking methods from other subsystems. The second subsystem is the *transformation manager*, which implements all the functions that apply a transformation on a PES. The transformation manager allows the user to select an appropriate transform and then applies it on the selected PES. Other important subsystems include the *undo manager* which performs inverse transformation operations on a PES instances, the *transform list manager* which keeps track of the transforms that have been applied on a PES instance, and a *modeling manager* that is responsible for constructing the semi-Markov chain model of a reduced PES instance. Each of these subsystems is discussed in detail.

5.4.1 The PES Manager

The PES manager is the heart of the CHITRA92 kernel. It performs the following functions.

1. **Mapping**

   We designed the system for PES's with between 500 to 1500 unique state vectors. These state vectors are mapped to unique identifiers called *program states.* Mapping the state vectors to program states ensures easy and rapid access to a state vector. The PES manager's first action upon encountering a PES is to convert the PES state vectors into program states. The mapping function(s) can specified by the user¶

¶At the time of this writing, the mapping function is a incrementer, which returns a value one greater than the value returned for the previous invocation of the function.

```
typedef struct unit
{
        int maxthreads; /* max components in the PES */
        int id; /* id of the PES instance */
        char binary_file[80]; /* Name of the PES */
        PSTRINGS pst_string; /* Pointer to the Value-Id table */
        PNODE pst_head; /* Pointer to the Mapper */

        char undo_file[80]; /* Backup file for Undo Manager */
        PPSNODE pst_oldpshead; /* Pointer to the Program State Table */
        PFILTER pst_oldfilstart; /* Pointer to the Filter List */

        char display_file[80]; /* Display File for the Unit */
        PPSNODE pst_pshead; /* Pointer to the Current Program State Table */
        PFILTER pst_filstart; /* Pointer to the Current Filter List */

        unsigned long maxtimestamp; /* Time stamp of Last state vector */
        int maxevents; /* Maximum Number of Events in the PES */
        int maxprogramstate; /* Next global program state */

        struct unit *pst_next;
}UNIT, * PUNIT;
```

Figure 5.4: Definition of a unit structure.

2. **Loading**

   The PES manager is responsible for loading a binary PES into the system. Once a PES has been loaded into the system, the user has an *instance* of the PES for analysis. The PES manager can load multiple instances of the same PES, to allow users to construct different models of a PES at the same time. No portion of the kernel ever modifies the actual PES file of a program.

   PES instances are implemented as *units* within the kernel. A unit is a structure that keeps track of all information associated with a PES instance. Figure 5.4 shows the definition of a unit structure. The kernel permits a PES instance to be constructed in three different ways.

   (1) The simplest and most common is the *map-and-load* where the PES file is mapped and loaded into the kernel.

   (2) The PES manager uses a pre-defined state vector-program state map to map the PES file and then loads it into the kernel.

   (3) The PES manager reads in a list of transforms along with the PES and applies each of the transforms in the list to the target PES.

3. **Other House-keeping functions**

   The PES manager is also responsible for keeping track of all PES instances in the kernel. An integer id is assigned to every active PES instance in a session and this id is used to distinguish between two instances of the same PES. All active units are kept in a list. When a new instance of a PES is created, its associated unit is added to the list and and when a PES instance is discarded from the system, its associated unit is removed from the list.

Every PES instance has a *display file* associated with it. The display file contains a set of records, where each record consists of the timestamp and the program state instead of the state vector it maps to. The display file is modified whenever a transform is applied, and is used to construct the semi-Markov chain model of the PES instance.

5.4.2 The Transformation Manager

CHITRA92's primary goal is to reduce a program execution sequence to model form through the successive application of user-selected transformations on a PES instance. Thus, the two primary software subsystems are the PES manager and the transformation manager. The transformation manager consists of several routines each of which takes a PES instance and a set of parameters as input and generates a new PES instance in place of the original. All of the transforms discussed in Chapter 3 have been implemented in the system.

1. **Filtering Transform**

   A filter aggregate transform can be applied to a PES instance both in the time domain and in the event domain. Filter aggregation is based on the premise that in most cases a program spends most of its time in a subset of the program states. Other program states are of interest merely because they are part of the transition sequence from one dominant program state to another. In the time domain, a filter aggregate transform takes as its input a PES instance and a threshold occurrence count value, and in the event domain, a filter aggregate transform takes as its input a PES instance and a threshold occurrence count value. All runs of program states in the instance, whose number of occurrences is less than the threshold, and have the same predecessor state and successor state are replaced by a composite state called the *filter aggregate state.* Similarly, in the time domain. a filter aggregate transform takes as its input a PES instance and a threshold occupancy time. All runs of program states in the instance, whose occupancy time is less than the threshold, and have the same predecessor state and successor state are replaced by a filter aggregate state.

   The unit associated with the instance keeps track of all the filter states in the PES instance. For each filter state, the predecessor dominant state, the successor dominant state, and the list of all sequences of filtered states that constitute the filter state and the number of occurrences of each sequence of filtered states are stored. Storing the number of occurrences of each filter state sequence is useful in computing the probability of a transition being made through that sequence.

The transformation manager also provides methods which the user can invoke to compute the occurrence count and the state occupancy duration for every program state in the PES. Figures 5.5 and 5.6 show system generated histograms of the occurrence counts and the occupancy times for a two-philosopher Dining Philosophers problem.

2. **Pattern Aggregation Transform**

   Aggregation is the process by which a sequence of program states are combined to form a new composite program state. Every occurrence of the same sequence of program states in the PES instance is also replaced by the composite state as a new global program state. An aggregation transform takes as its input a PES instance and a sequence of global program states called a *pattern*, and modifies the PES instance such that every occurrence of the pattern in the PES instance is replaced by a new program state called the *pattern aggregate state.*

   The aggregation transform has been implemented using the KMP fast pattern-matching algorithm [47, 48]. Subsequences within a PES can be identified and tagged in linear time.

   Three stages of aggregation are illustrated: before pattern matching (Figure 5.7) matching, after pattern matching (Figure 5.8), and aggregating the patterns in the PES instance (Figure 5.9).

3. **Clipping Transform**

   Clipping is the process by which transient portions (initial and final) of a PES are removed. A clipping transform takes as its input a PES instance and a bounding region specified either as beginning timestamp and a ending timestamp (in the time

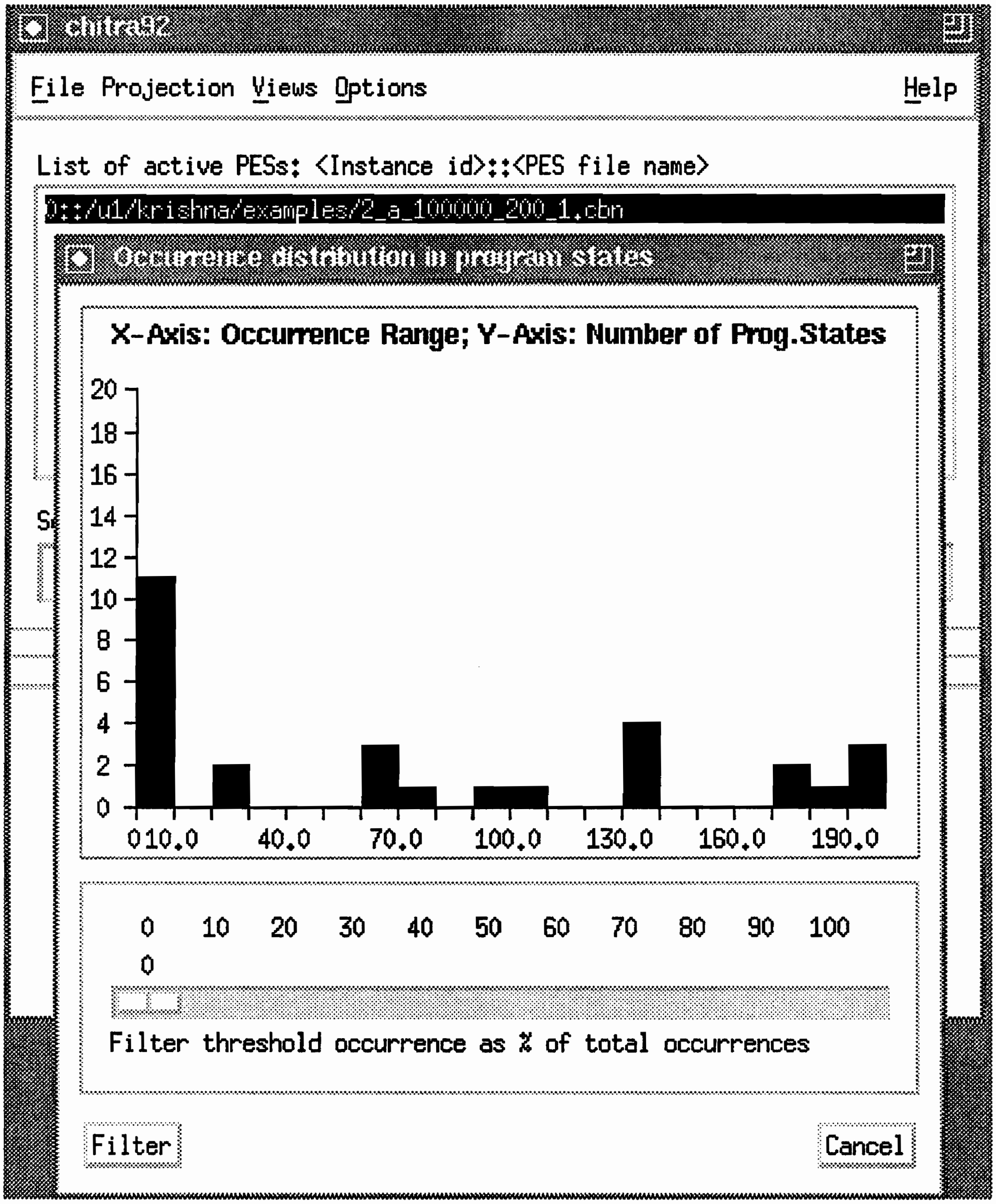
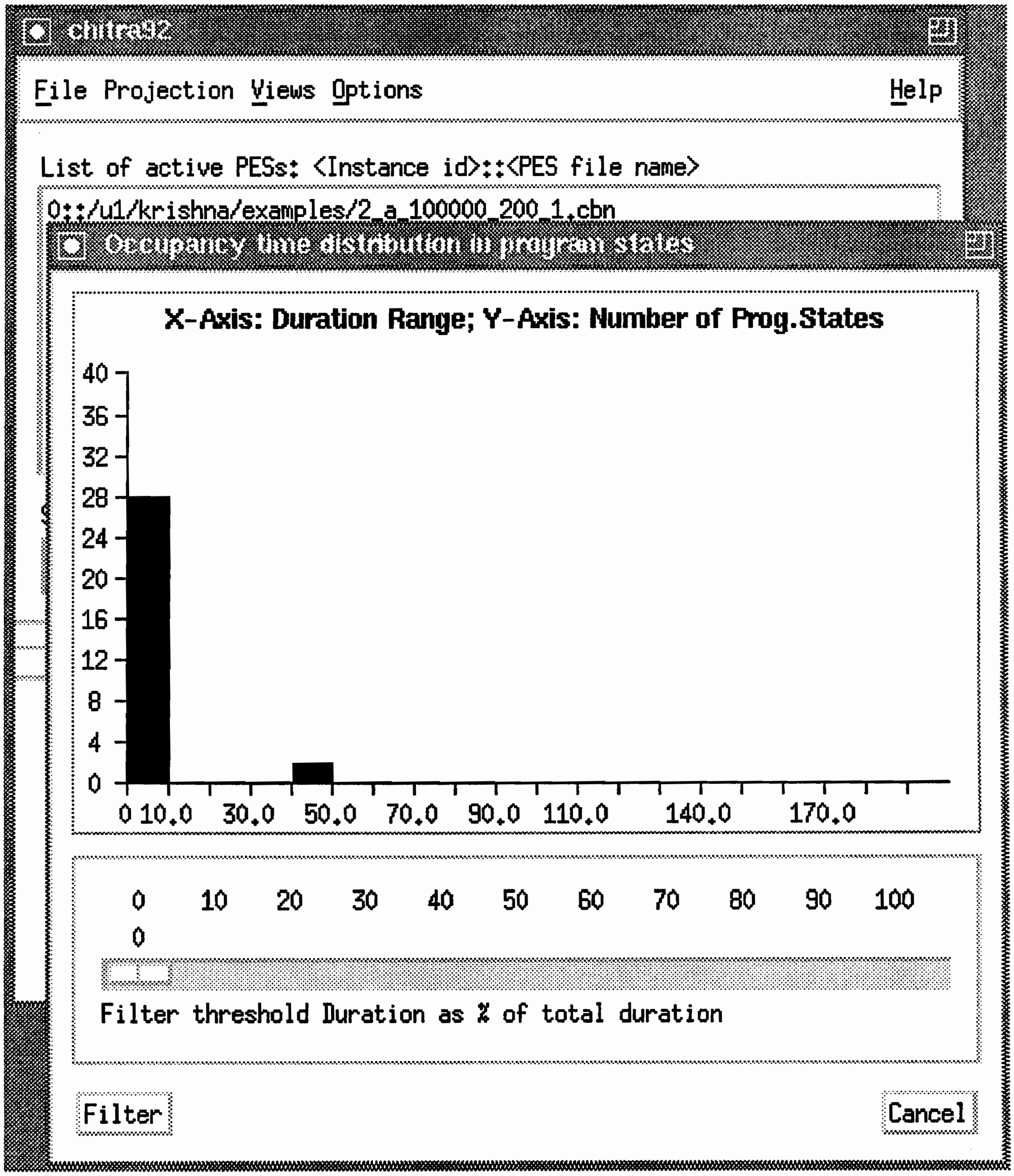

chitra92
File Projection Views Options
Help
List of active PESs: <Instance id>::<PES file name>
0::/u1/krishna/examples/2_a_100000_200_1.cbn
Occurrence distribution in program states
X-Axis: Occurrence Range; Y-Axis: Number of Prog.States
20
18
16
14
12
10
8
6
4
2
0
0 10.0
40.0
70.0
100.0
130.0
160.0
190.0
0 10 20 30 40 50 60 70 80 90 100
0
Filter threshold occurrence as % of total occurrences
Filter
Cancel

Figure 5.5: Occurrence distribution of program states for a two-philosopher Dining Philosophers problem.


chitra92
File Projection Views Options Help
Occupancy time distribution in program states
Filter
Cancel

List of active PESs: <Instance id>::<PES file name>

0::/u1/krishna/examples/2_a_100000_200_1.cbn

X-Axis: Duration Range; Y-Axis: Number of Prog.States

Filter threshold Duration as % of total duration

Figure 5.6: Duration distribution of program states for a two-philosopher Dining Philosophers problem.

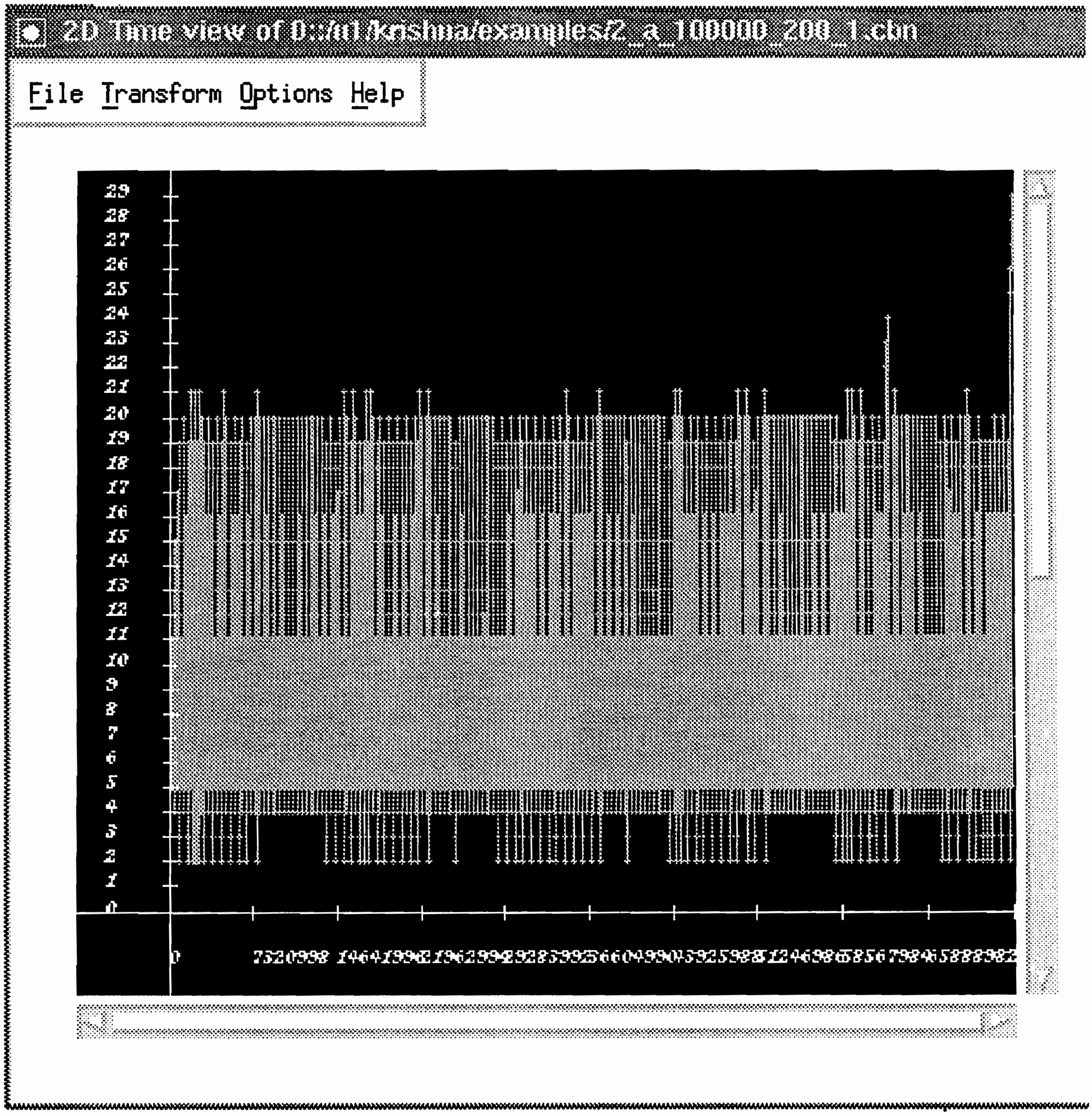

2D Time view of 0::/u1/krishna/examples/2_a_100000_200_1.cbn
File Transform Options Help
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0
7320998

Figure 5.7: 2D time view of a two-philosopher Dining Philosophers problem before pattern matching.

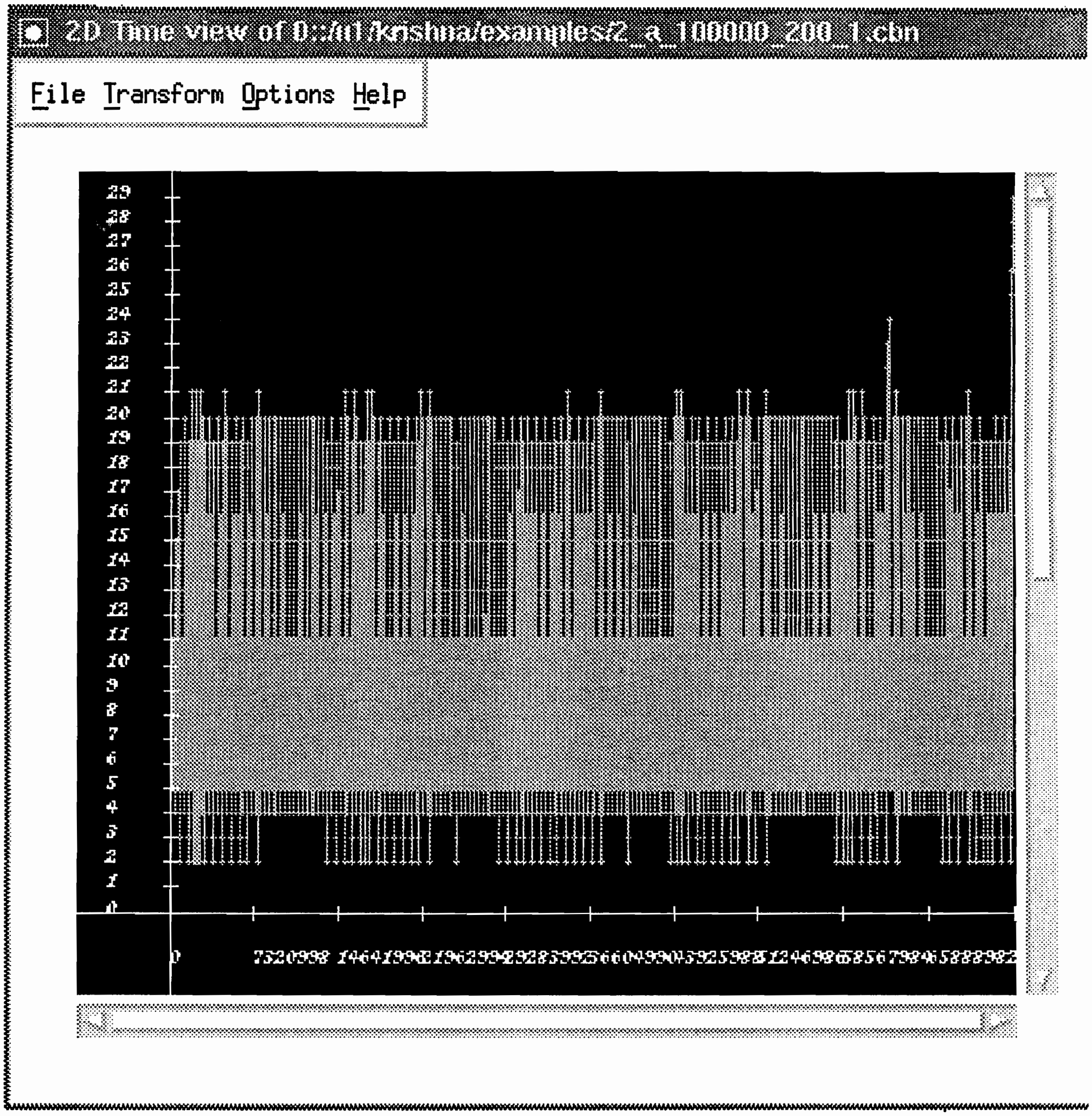

2D Time view of 0::/u1/krishna/examples/2_a_100000_200_1.cbn
File Transform Options Help
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0

Figure 5.8: 2D time view of a two-philosopher Dining Philosophers problem after pattern matching.

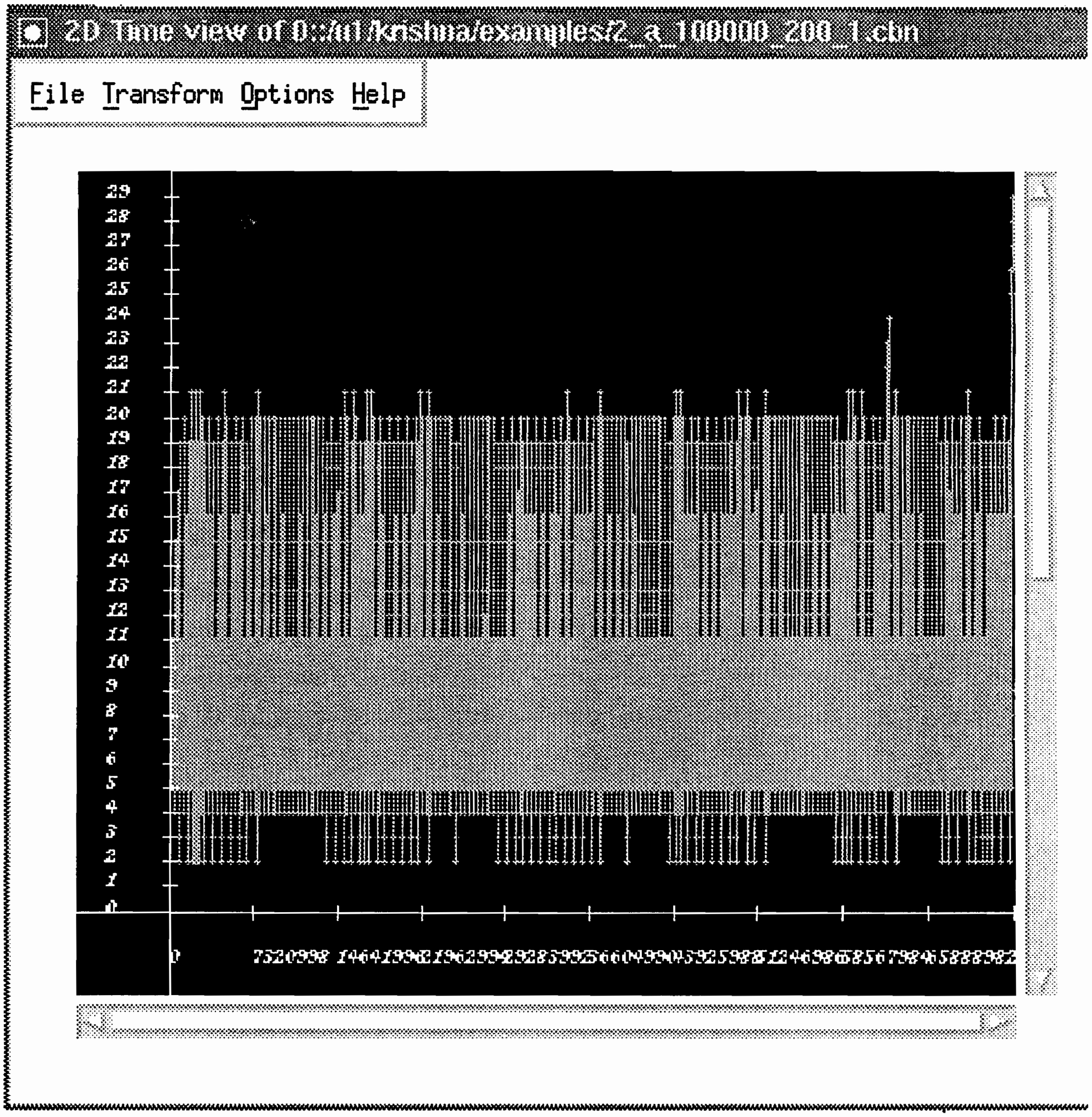

2D Time view of /n1/krishna/examples/2_a_100000_200_1.ctm
File Transform Options Help
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0

Figure 5.9: 2D time view of a two-philosopher Dining Philosophers problem after aggregation.

domain) or a beginning event id and an ending event id (in the event domain) and modifies the PES such that those portions of the instance that lie outside the bounding region are removed from the PES instance.

4. **Projection Transform(s)** Projections are special-purpose transforms that a user may decide to incorporate into the CHITRA92 system. An example of a projection is a transform that collapses the state vector of a PES by removing one or more components The CHITRA92 system provides a facility by which the user can define new transformation modules and link them to the system.

5.4.3 The Undo Manager

Undoing is the process by which a previously applied transform is undone and the PES instance existing before the application of the transform is restored. If a transformation can be thought of as a function that takes as its input a PES and a set of input parameters and returns as its output a new PES, then the undo manager provides the corresponding inverse function for the transformation. Undoing is a useful feature for the kernel to provide because it facilitates an iterative approach to transforming a PES instance. If a user is dissatisfied with the result of a transformation on a PES instance, then the corresponding undo transform function can be invoked to restore the PES instance to its previous state.

All inverse functions to the previously discussed transformations have been provided. These include the undo operation for filtering, aggregation, clipping and projections.

The undo manager works by making a backup of all information that is contained in a unit before applying a transformation. Once the transformation is complete, if the user desires, the backup information of the unit can be copied into the unit by the undo manager to restore the original PES.

5.4.4 The Transform List Manager

The CHITRA92 kernel keeps track of all the transformations applied on a PES starting from the time it was loaded into the system. The transform list manager provides functions by which new transforms can be appended to the transform list or removed from the list. The transform list manager performs two important functions:

1. **Saving a modified PES instance**

   The resolution of a PES into model form is accomplished through the application of several transforms. This is a trial-and-error process and make take several sessions. It is necessary to save the modified PES instance across work sessions.

   The kernel does not save the display file itself between sessions. Instead, the transform list until that point is written to disk. To restore a PES instance to its last modified state, the PES manager loads the PES file set and the transform list file. Each of the transforms are applied successively (automatically while the user waits), until the saved state of the PES instance is obtained.

2. **Multi-level Undo** The kernel undo manager currently supports only a *one-level* undo, meaning that only the effects of the most recently applied transform can be undone. The undo manager and the transform list manager can interact together to provide a multi-level undo, where the effects of more than one transform can be undone.

   If a modified PES instance has to be undone over several transformations, then the transform list manager can extract each transform starting from the first and successively apply them to a new instance of the original PES. This process can be repeated until the required PES instance is obtained.

5.4.5 The Modeling Manager

The modeling manager is responsible for generating a continuous-time, homogeneous semi-Markov chain model of a PES instance. For a PES instance the following information is generated:

1. A state transition matrix where the rows and columns correspond to unique program states in a PES instance. Every entry, (i,j), in the state transition matrix is a probability estimate of a transition being made from the program state corresponding to the ith row in the state transition matrix to the jth column in the matrix.

2. For every unique program state of the PES instance, the following information is computed:

   - The program state id and the state vector that the program state maps to in the PES. If the program state is a pattern aggregate state o a filter aggregate state, then more information such as the subsequence of program states that comprises the composite state is determined.
   - The number of occurrences of the program state in the PES instance
   - The total occupancy time of the program state in the PES instance.
   - The mean occupancy time, the minimum occupancy time, the maximum occupancy time and the standard deviation of the occupancy time for the program state.

5.5 The CHITRA92 Visualization System

The final component of the CHITRA92 system is the visualization system. This system forms the front-end to the entire performance analysis system and is the largest in terms of total lines of code.

5.5.1 Visualization Software Overview

At the lowest level, the visualization system comprises of several graphical views of PES instances. Examples of views are barcharts, histograms, X-Y graphs, text dumps, and planar graphs.

The visualization system has been implemented as a X11 application. It uses the OSF/Motif widget set, the Athena widget set and the Xt toolkit. The visualization system supports color and grey scale terminals, although it is preferable to use a color workstation for the system.

The top level interface of the CHITRA92 system is shown in Figure 5.10. It consists of three display regions: the root menu system, the PES instance list, and the message area. The root menu system consists of four main menu items: $\boxed{\text{File}}$, $\boxed{\text{Projection}}$, $\boxed{\text{Views}}$, and $\boxed{\text{Options}}$.

The $\boxed{\text{File}}$ menu item contains the following items:

1. $\boxed{\text{Load PES}}$ allows the user to perform one of three functions: loads a text CSL, via translation of the CSL specification into a binary representation of the PES, load an instance of the PES binary file; and load an instance of PES via a saved transform list file.

2. $\boxed{\text{Close selected PES}}$ allows the user to close an instance of a PES and free all system resources associated with it.

3. $\boxed{\text{Save selected PES}}$ saves the transform list associated with an instance of a PES to a file.

4. $\boxed{\text{About selected PES}}$ provides statistics such as the name of the PES instance, the number of components in the instance the name of the display file, the maximum number of program states, the cumulative number of events in the instance and the total duration of the instance.

5. $\boxed{\text{About Chitra}}$ provides information about the CHITRA92 system.

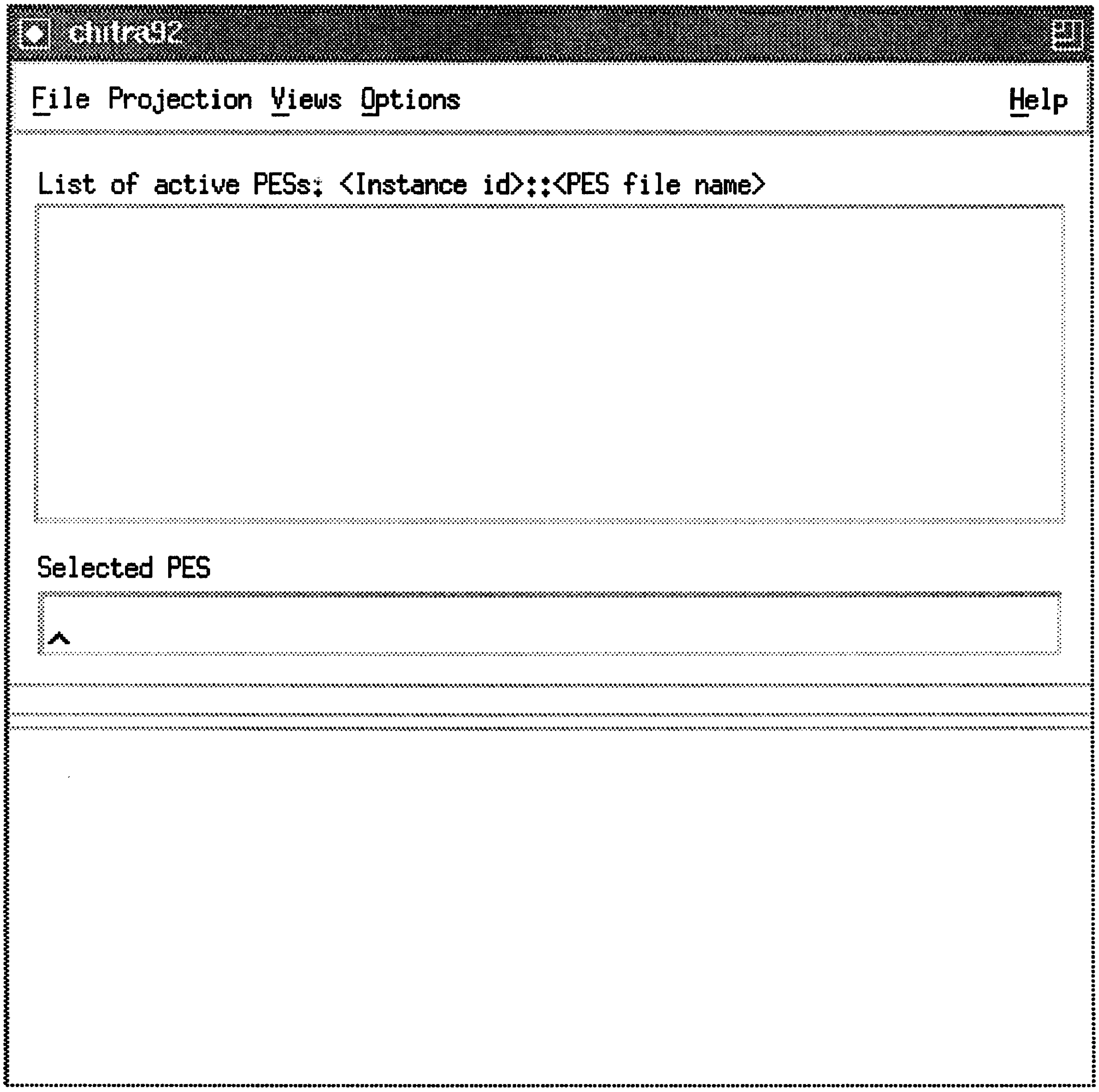

chitra92
File Projection Views Options
Help
List of active PESs: <Instance id>::<PES file name>
Selected PES

Figure 5.10: Top-level interface for the CHITRA92 system.

6. $\boxed{\text{Exit}}$ terminates execution of the current Chitra92 session.

The $\boxed{\text{Projection}}$ menu item provides a list of all available user defined transforms offered by the system. If a user selects a particular transform, a dialog box allows the user to enter the parameters to be provided for the transform.

The $\boxed{\text{Views}}$ menu item allows the user to construct a graphical display of the selected PES. CHITRA92 provides 2D X-Y displays, periodogram displays, text displays and state transition matrix displays.

The $\boxed{\text{Options}}$ allows the user to retrieve information about the currently selected PES instance such as pattern aggregate state information, filter aggregate state information, and the transformation list of the PES instance.

The second display area is a list box which displays the currently available PES instances that the user can select and work on. When a user loads a PES instance into the system, the name of the PES instance appears in the list box. Similarly, when the user closes a PES instance, the name of the PES instance is removed from the list box.

The third display area is the message area where system related information is presented.

The six types of views can be invoked on an active PES instance or when a PES is loaded into the system for the first time. Each of the views have the following menu items:

1. $\boxed{\text{File}}$ which allows the user to refresh the view, query the system for information about the PES instance being viewed, and close the view.

2. $\boxed{\text{Transform}}$ which allows the user to select and apply a transform on the PES instance being viewed. The undo option can also be selected from this menuitem.

3. $\boxed{\text{Options}}$ which allows the user to select various options which modify the views settings. For example, the line style in a 2D X-Y view can be changed and the X-Y axes can be displayed or hidden.

5.6 Summary

The important features of the CHITRA92 architecture are as follows:

1. The primary goals of the CHITRA92 system are that it be portable across multiple platforms, be scalable, easily extensible and versatile enough to support performance data from different architectures.

2. The PES description language CSL allows performance data to be ported across a variety of platforms because the CSL specifications are ASCII files that are easily portable.

3. The translator output consisting of two files derived from a single CSL specification file solves two problems. The ASCII setup file allows a user to edit the setup file, if required, without regenerating the binary trace file. The binary trace file allows for compactness in the data representation and fast and efficient access.

4. The CHITRA92 kernel consists of several interacting subsystems: the PES manager manages multiple instances of PES's and is responsible for loading a PES into the system; the transformations manager provides routines that can invoked to transform a PES and the modeling manager is responsible for generating the semi-Markov chain model of a PES instance.

5. The CHITRA92 visualization system provides the top-level interface to the system and access to the PES instances via views. Views provide interfaces through which the user can apply a transformation on a PES instance. The views supported include: 2D X-Y time views, 2D X-Y event views, Text views, FFT (time) views, FFT (event) views and semi-Markov model views.

Chapter 6

Conclusions and Future Research

6.1 Summing up CHITRA92

The goal of this thesis was to design and implement a performance analysis environment, that allows a user to model parallel program behavior as semi-Markov chains. We have successfully implemented the CHITRA92 Performance Analysis Environment on DECStations using the X11 Window System and the OSF/Motif 1.1 toolkit.

We summarize the important and unique contributions of the CHITRA92 performance analysis environment.

1. **Model generation**

   Visualization systems have become prominent in performance analysis studies of parallel programs. Our contention is that while visualization systems are very useful in exploiting perceptive and intuitive abilities of humans to comprehend large volumes of data, they really do not help in the analysis. The notion of generating a model of the input PES is a novel approach. Modeling makes two significant contributions to the study of PES's. First, models are valuable tools in predicting PES behavior. The usefulness of the model is based on how accurate its predictions are for different input parameter sets. The second contribution is subtler. While visualization assists in increasing the ability of humans to comprehend large volumes of data, modeling performs the converse. A model of a large PES captures all of the characteristics of the PES but its representation is concise. The understanding obtained from the model however is the same or more likely better than that obtained from examining the data itself.

CHITRA92 generates a semi-Markov model of an input PES. The model provides statistics such as maximum and minimum occupancy times of a program state, the mean occupancy time and the number of occurrences of the program state. The transition frequencies from one program state to the next are also computed.

2. **Transformations**

   The CHITRA92 system provides four very powerful transformations which assist a user in reducing a PES so as to make it easy for semi-Markov chain model construction. Pattern aggregation allows the identification of determinism within an PES. Filter aggregation allows the identification and elimination of randomly occurring subsequences, and/or possible noise signals. The clipping transform allows the user to select the steady-state portion of the system for analysis. Finally, the system's design allows user-defined transforms (projections) to be easily integrated into the system.

3. **Studying periodicity in PES's**

   The system treats any PES as a time-series data set and applies spectral analysis techniques to determine the existence of periodic behavior in a PES. The discrete Fourier transform of the time-series data is computed and a periodogram of the DFTS is generated. The occurrence of spikes within the periodogram indicate a concentration of energy, which in turn implies that the maximum possible periodicity is exhibited at the frequency of the spike. To our knowledge, no existing system performs this kind of analysis.

4. **Visualization of PES's**

   Most existing performance analysis environments use visualization as a solution to the performance analysis problem. We believe that such an assumption is wrong. Visualization systems go a long way towards raising the upper bound on data that human beings can intuitively comprehend (this is clearly illustrated by the fact that graphical displays can present greater amounts of information that can be comprehended by a human as compared to textual dumps). Analysis (exemplified by modeling) is something that humans cannot intuitively perform. Visualization therefore is not an end to itself. It can be used effectively to assist in the analysis of the problem, which is how it is in the CHITRA92 performance analysis environment. 2D X-Y views, Text views, Fourier views, and Semi-Markov views can be used effectively to generate an accurate model of a PES. The system also allows a user to load multiple instances of PES's and have multiple views of a single instance of a PES.

5. **PES description language**

   The biggest concern in the design of the system is that it be easy to use and widely available. CHITRA92's design goals of portability, scalability and versatility all reflect this concern. The solution offered by the system is that it provides a PES description language (CSL). While portability of the system itself has been ensured by using standard and widely available software development tools, PES's have been made portable by allowing two representation formats – text (in the form of a CSL specification) and binary. The text representation ensures portability and the binary ensures efficient access and storage of a PES.

One question that has been raised is to what kind of an audience is the CHITRA92 system targeted to. Most software developers are intimately acquainted with the underlying architecture as well as the system software of their development platform. Novice users or intermediate-level users typically work with pre-existing applications. They are usually interested in the results and not in the performance of the software, much less in optimizing its performance. CHITRA92 is performance analysis system for programs and, consequently, is targeted towards software developers.

6.2 Beyond CHITRA92

We have successfully completed two generations of systems in the Chitra project – CHITRA91 and CHITRA92. Work is already underway for the third generation of the Chitra project. CHITRA93 is expected to be released by the end of 1993. A variety of both models as well as features are being planned for CHITRA93. We have broadly classified further research into three classes. Firstly, semi-Markov chains, while sufficient for some classes of resource-sharing, cyclic programs, cannot model most other classes of programs. The study of alternate models is therefore very important. The second area of research is in extension of the architecture of the CHITRA92 system. For example, synchronized views of PES's is a useful feature for examining multiple execution runs of the same program. From enhancing the existing implementation of the system, a number of research issues arise. The third important area of research is to study a greater variety of applications. Our knowledge of the behavior of programs can increase only if a wider range of applications are studied.

We discuss each of these research areas briefly.

6.2.1 Implementation Enhancements

There are number of enhancements that can be made to the system. First, a greater variety of views can be incorporated into the visualization system. Two useful views are a planar graph layout of the semi-Markov model and synchronized 2D X-Y view of multiple instances of PES's. Secondly, the transformation subsystem can be enhanced by adding more transformations as the need arises. For example, as of now projections are applied to a unmodified PES and inorder to apply them to modified PES's, we need to decide how the projection will treat existing aggregate states in the modified PES. Thirdly, greater support is required for the PES representation both within the CSL translator and in the binary and text files. Finally, for the system to be scalable, a parallel version of the system should be developed.

6.2.2 Alternate Models

Four new possible models for PES's are currently being studied. The first is to use decision trees to identify determinism in a PES. Decision trees determine the influence that a previous program state has in the PES making a particular transition. The second is to model each component of a state vector independently and create a composite model by gluing together the individual models appropriately. The individual models could be semi-Markov chains themselves. The third modeling strategy is to compute rules by which deterministic subsequences within the PES could be predicted. The last modeling technique is to model a PES as a quasi-stationary stochastic process.

6.2.3 Target Applications

A large number of real world applications ranging from problems in operating systems to large-scale scientific and numeric applications are being studied. Some of the more interesting problem domains are described below:

1. **Transaction processing**

   Performance data traces have been obtained from instrumented versions of IBM's order placement system, developed in CICS. Efforts are being made to modify the trace data obtained so that it is suitable for analysis by the CHITRA92 system.

2. **Numeric and scientific applications**

   Two well-known numerical analysis problems - gaussian elimination and large size matrix multiplications, are being used to study the possibility of predicting the existence of deterministic subsequences within a PES. Another interesting scientific problem being studied is the simulation of the dynamic response of a gas turbine compression system operating under stable, unstable and reverse flow conditions. This particular application has been parallelized to run on a 4-node transputer bed [55].

3. **Communication Protocols**

   CHITRA92 has already been used on data generated from an instrumented TCP/IP package developed for IBM PC-compatibles. This study will be continued during 1993. The goal here is to determine what specific factor limits the performance of the TCP/IP suite [8, 3]. Two other communication protocols are being studied. The first is a study to tune the performance of an MPEG player from the University of California at Berkeley, and the second is to assist in the design and development of NMFS, a network multimedia file system protocol.

4. **Parallel discrete-event simulation**

   Two studies are being undertaken in the area of discrete-event simulation [24]. The first is to model the behavior of Lubachevsky's bounded lag problem [34, 35] and the second is to study the quality of native code automatically generated from a parallel simulation program specification.

5. **Operating Systems**

   The Chitra project was initiated with a study of the Dining Philosophers problem [16, 7, 3]. More examples of the Dining Philosophers problem are being studied as well as variations on the Dining Philosophers such as the Drinking Philosophers problem.

Performance diagnosis for parallel programs still remains an open problem. Parallel programming will not enter the mainstream of computing unless there are tools that can help identify bottlenecks in parallel program source code. Constructing models of program behavior is a promising approach to this problem. It is our hope that the Chitra project eventually provides a collection of tools and services that can do this.

Appendix A

Chitra Specification Language Grammar

The following is the grammar for the Chitra Specification Language.

$$
\begin{aligned}
\langle program\rangle &\rightarrow \langle program\_name\rangle; \langle block\rangle .EOF \\
\langle program\_name\rangle &\rightarrow PROGRAM\ IDENTIFIER \\
\langle block\rangle &\rightarrow BEGIN \langle setup\rangle \langle trace\rangle END \\
\langle setup\rangle &\rightarrow SETUP : \langle maxcomponents\rangle \langle component\_list\rangle \langle thread\_list\rangle \\
\langle maxcomponents\rangle &\rightarrow MAXCOMPONENTS = INTEGER \\
\langle component\_list\rangle &\rightarrow \langle component\rangle \langle component\_list\rangle \\
&\quad\ \mid \langle component\rangle \\
\langle component\rangle &\rightarrow COMPONENT \langle component\_name\rangle \langle state\_names\rangle \\
\langle component\_name\rangle &\rightarrow \langle id\_range\rangle = STRINGCONST, INTEGER; \\
\langle id\_range\rangle &\rightarrow \langle id\rangle , \langle id\_range\rangle \\
&\quad\ \mid \langle id\rangle \\
\langle id\rangle &\rightarrow INTEGER - INTEGER \\
&\quad\ \mid INTEGER \\
\langle state\_names\rangle &\rightarrow \langle statename\rangle ; \langle state\_names\rangle \\
&\quad\ \mid \langle statename\rangle \\
\langle statename\rangle &\rightarrow INTEGER = IDENTIFIER \\
\langle trace\rangle &\rightarrow TRACE : \langle trace\_component\_list\rangle
\end{aligned}
$$

$$\begin{aligned}
\langle trace\_component\_list\rangle &\rightarrow \langle trace\_component\rangle\langle trace\_component\_list\rangle \\
&\quad \mid \langle trace\_component\rangle \\
\langle trace\_component\rangle &\rightarrow \langle time\_stamp\rangle\langle statevector\rangle \\
\langle time\_stamp\rangle &\rightarrow INTEGER \\
\langle state\_vector\rangle &\rightarrow \langle integer\_list\rangle \\
\langle integer\_list\rangle &\rightarrow INTEGER, \langle integer\_list\rangle \\
&\quad \mid INTEGER
\end{aligned}$$

Appendix B

Notes on the CHITRA92 Source Code

1. Tim Lee was responsible for the design and implementation of the model view in the system.
2. The backend for theCSL translator is currently under development by Horacio Cadiz.
3. The FFT routines are from the book: *Numerical Methods and Software* by David Kahaner, Cleve Moler and Stephen Nash.
4. The histogram widget is from the Free Widget Foundation, coordinated by Brian Totty, Department of Computer Science, University of Illinois, Urbana-Champaign.

Wherever appropriate within the code, copyright notices have been retained to acknowledge these contributions.

REFERENCES

[1] Gordon Bell, "Ultracomputers, A Teraflop Before Its Time," *Communications of the ACM*, Vol 29, No. 12, Aug. 1992, pp. 27-47.

[2] W. D. Hillis and G. L. Steele Jr., "Data Parallel Algorithms," *Communications of the ACM*, Vol 35, No. 8, Dec. 1986, pp 1170-1201.

[3] M. Abrams, N. Doraswamy, and A. Mathur, "Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event, and Frequency Domains," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 6, pp 672-684, November 1992.

[4] M. Abrams and A. K. Agrawala, "Performance Study of Distributed Resource Sharing Algorithms," *IEEE Dist. Processing Technical Committee Newsletter*, Vol. 7, No. 3, pp. 18-26, Nov. 1985.

[5] M. Abrams, "Design of a Measurement Instrument for Distributed Systems," Research Report RZ1639, IBM Research Division, Zurich Research Laboratory, 1987.

[6] M. Abrams and A. K. Agrawala, "Geometric Performance Analysis of Mutual Exclusion," Department of Computer Science, Virginia Tech, TR 90-58, 1990.

[7] M. Abrams and N. Doraswamy, "An Introduction to Visualizing Program Execution Dynamics with Chitra," Department of Computer Science, Virginia Tech, TR 92-23, May 1992.

[8] M. Abrams and Q. Chen, "A New View on What Limits TCP/IP Throughput in Local Area Networks," Department of Computer Science, Virginia Tech, TR 91-23, 1991, submitted for publication.

[9] M. Abrams, Diagnosing Parallel and Distributed Software Performance Problems," *National Science Foundation Innovative Grant Proposal*, Department of Computer Science, Virginia Tech, November 1991.

[10] G. Balbo, G. Chiola, S. C. Bruell, and P. Chen, "An Example of Modeling and Evaluation of a Concurrent Program Using Colored Stochastic Petri Nets: Lamport's Fast Mutual Exclusion Algorithm," *IEEE Trans. on Parallel and Distributed Systems 3*, pp. 221-240, March 1992.

[11] B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Presto: A System for Object-Oriented Parallel Programming," TR 87-09-01, Dept. of Computer Science, Univ. or Washington, Sept. 1987.

[12] U. N. Bhat, *Elements of Applied Stochastic Processes*, 2nd. ed, New York: John Wiley, 1984, pp. 290-294.

[13] M. H. Brown, *Algorithm Animation*, MIT Press, Cambridge, MA, Ph.D. thesis, Department of Computer Science, Brown University, 1988.

[14] S.D. Carson and P. F. Reynolds Jr., "The Geometry of Semaphore Programs," *ACM TOPLAS*,Vol. 9, No. 1, Jan. 1987.

[15] Jack Dongarra, Orlie Brewer, James Arthur Kohl, and Samuel Fineberg, "A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors," *Journal of Parallel and Distributed Computing* , Vol. 9, No. 2, pp. 185-202, June 1990.

[16] N. Doraswamy, *Chitra: A Visualization System to Analyze the Dynamics of Parallel Programs*, M.S. thesis, Dept. of Computer Science, Virginia Tech, Dec. 1991.

[17] A. Duda, G. Harrus, Y. Haddad, G. Bernard, "Estimating Global Time in Distributed Systems," *Proc. 7th Int. Conf. on Distributed Computing Systems,* Berlin, pp. 299-306, Sept. 1987.

[18] Robert J. Fowler, Thomas J. LeBlanc and John M. Mellor-Crummey, "An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors," *Proceedings of Workshop on Parallel and Distributed Debugging*, ACM SIGPLAN Notices Vol. 24, No. 1, pp. 163-73, Jan. 1989.

[19] M. A. Holliday and M. K. Vernon, "A Generalized Timed Petri Net Model for Performance Analysis," in *Proc. Int. Workshop on Timed Petri Nets*, Cat. No. 2187-3, pp. 181-90, Turin, July 1985.

[20] Alfred A. Hough and Janice E. Cuny, "Belvedere: Prototype of a Pattern-oriented Debugger for Highly Parallel Computation," *Proc. of the 1987 Intl. Conf. on Parallel Processing*, pp. 735-8, 1987.

[21] Thomas J. LeBlanc, John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers*, Vol. C-36, No. 4, pp. 471-82, April 1987.

[22] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz, The Pablo Performance Analysis Environment," submitted for publication, August 1992.

[23] Allen D. Malony, D. A. Reed, and H. A. G. Wijshoff, Performance Measurement Intrusion and Perturbation Analysis," *IEEE Trans. on Parallel and Distributed Systems*, Vol 3, No. 4, July 1992.

[24] Anup Mathur and Marc Abrams, Toward a Machine Assisted Software Performance Diagnosis Methodology," Department of Computer Science, Virginia Tech, TR 93-12, April 1993.

[25] Blaine Gaither, "Instrumentation for Future Parallel Systems," in *Instrumentation for Future Parallel Systems*, ed. M. Simmons and R. Koskela, ACM Press, pp. 111-120, 1989.

[26] Allen D. Malony, "JED: Just an Event Display," in *Performance Instrumentation and Visualization*, ed. M. Simmons and R. Koskela, ACM Press, pp. 99-114, 1989.

[27] Allen D. Malony, *Performance Observability*, Ph.D. thesis, Computer Science Dept., Univ. of Illinois, Aug. 1990.

[28] Allen D. Malony and Daniel A. Reed, "Visualizing Parallel Computer System Performance," CSRD Tech. Report No. 812, Computer Science Dept., Univ. of Illinois, March 1988.

[29] Margaret Martonosi, Anoop Gupta, and Thomas Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs," *Proc. ACM SIGMETRICS*, Newport, RI, pp. 1-12, June 1992.

[30] Ted Lehr, Zary Segall et. al., "Visualizing Performance Debugging," *IEEE Computer*, pp 38-51, October 1989.

[31] Aaron J. Goldberg and John L. Hennessy, "Mtool: An Integrated System for Performance Debugging Shared memory Multiprocessor Applications,"*IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 1, pp 28-40, January 1993.

[32] Bershad, B.N., Lazowska, E.D., and Levy, H.M, "PRESTO: A system for Object-Oriented Parallel Programming," *Technical Report 87-09-01.*, Department of Computer Science, University of Washington, Seattle, Washington, Jan. 1988.

[33] Gary Miller and Kathy Walrath, *NeXT Operating System Software*, NeXT Computer, 1990.

[34] Lubachevsky, B., "Efficient distributed event-driven simulations of multiple loop networks," *Comm. ACM.,* vol. 32, no. 1,111-123, Jan. 1989.

[35] Lubachevsky, B., "Synchronization Barrier and Related Tools for Shared Memory Parallel Programming," *International Journal of Parallel Programming,* vol. 19, no. 3,226-250, July. 1990.

[36] Charles E. McDowell and David P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys*, Vol. 21, No.4, pp. 593-622, Dec. 1989.

[37] B. Miller and C. Q. Yang, IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs," in *Proc. of the 7th International Conference on Distributed Computing Systems*, September 1987.

[38] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, pp. 206-17, April 1990.

[39] William C. Brantley and Henry Y. Chang, "Support Environment for RP3 Performance Monitor," in *Performance Instrumentation and Visualization*, ed. M. Simmons and R. Koskela, ACM Press, pp. 99-114, 1989.

[40] D. Wybranietz and D. Haban, Monitoring and Measuring Distributed Systems," in *Performance Instrumentation and Visualization*, ed. M. Simmons and R. Koskela, ACM Press, pp. 27-46, 1989.

[41] Carlo Ghezzi and Mehdi Jazayeri, *Programming language concepts*, 2nd. ed, New York: John Wiley, 1987.

[42] L. Kleinrock, *Queueing Systems Vol. I: Theory*, 2nd. ed, New York: John Wiley, 1984, pp. 290-294.

[43] B. Plateau, "On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms," *Proc. SIGMETRICS*, Austin, pp. 147-154, Aug. 1985.

[44] Sekhar R. Sarukkai. "Performance Visualization and Prediction of Parallel Supercomputer Programs: An Interim Report," TR 318, Computer Science Dept., Indiana Univ., Nov. 1990.

[45] David Socha, Mary L. Bailey, and David Notkin, "Voyeur: Graphical Views of Parallel Programs," *in Proc. of Workshop on Parallel and Distributed Debugging*, ACM SIGPLAN Notices, Vol. 24, No. 1, pp. 206-15, Jan. 1989.

[46] Allan M. Tuchman, Michael W. Berry, "Matrix Visualization in Design of Numerical Algorithms," *ORSA Journal on Computing*, Vol. 2, No. 1, pp. 84-92, Winter 1990.

[47] D. E. Knuth, J. H. Morris, V. R. Pratt, Fast pattern matching in strings," *SIAM Journal on Computing*, Vol. 6, pp 323-350, June 1977.

[48] Udi Manber, *Introduction to Algorithms: A Creative Approach*, Addison Wesley Publishing Company, Reading, Mass., 1989.

[49] Open Software Foundation, *OSF/Motif Programmer's Reference Manual*, Prentice Hall, Englewood Cliffs, NJ

[50] Doug McMinds, *Mastering OSF/Motif Widgets,* Addison Wesley Publishing Company, Reading, Mass., 1992.

[51] Adrian Nye, *X toolkit intrinsics programming manual*, O'Reilly and Associates, 1990.

[52] Adrian Nye, *X toolkit intrinsics reference manual*, O'Reilly and Associates, 1990.

[53] Adrian Nye, *Xlib programming manual*, O'Reilly and Associates, 1990.

[54] Adrian Nye, *Xlib reference manual*, O'Reilly and Associates, 1990.

[55] John K. Harvell, Walter F. O'Brien, "Computational Considerations associated with the development of Near-Real-Time Dynamic Simnulations for Propulsion Applications," *in Proceedings of the 30th Aerospace Sciences Meeting and Exhibit*, AIAA 92-0560, January 1992.

VITA

Krishna Ganugapati was born on June 3, 1966 in Manchester, United Kingdom. He spent his early childhood in the UK and in Australia. He received a Bachelor's degree in Computer Science and Engineering with distinction from the University of Mysore, India in 1988. From November 1988 to July 1990, he worked as a Software Design Engineer in PSI Data Systems, India where he had the opportunity to write software for Groupe Bull's peripherals division in Belfort, France. He received the MS degree in Computer Science from the Department of Computer Science, Virginia Polytechnic Institute and State University in May 1993.

Immediate plans for Krishna include a position with Microsoft Corporation, Redmond, Washington.