(2)

1 Numerical Productivity Measures

K.T.Narayana
Department of Computer Science
Whitmore Laboratory
Pennsylvania State University
University Park, Pa16802
Phone:(814)-863-0147
narayana@shire.cs.psu.edu

Grant Titile: Real-Time System Specification and Verification
Grant Number: N00014-89-J-1171
Reporting Period: Oct 1,1988-Sep 30,1988.

Published Papers: 1
Papers Submitted for Publication: 4
Technical Reports: 6
Graduate Students Supported: Eric Shade from Jan,1 1989-Aug 1989 (half time)
Principal Investigator : Summer 1989 and 10% Academic.

K.T.Narayana
Department of Computer Science
Whitmore Laboratory
Pennsylvania State University
University Park, Pa16802
Phone:(814)-863-0147
narayana@shire.cs.psu.edu
Grant Titile: Real-Time System Specification and Verification
Grant Number: N00014-89-J-1171
Reporting Period: Oct 1,1988-Sep 30,1988.

2 Summary of Technical Progress

We have developed a formal semantic model for real-time concurrency under limited parallelism. The model addresses memory access mechanisms, limited parallelism under asynchronous processors. In the framework of the model, various scheduling paradigms can be imposed.

We formulated a language concept of tri-sections. The concept combines nondeterministic multiway synchronization of processes and processor holding into a single primitive construct. The use of the concept has been demonstrated with a process control system, resource allocation problems, and elevator systems. The concept allows the construction of maximally parallel regions in an otherwise limited parallel execution model. In a semantic sense, the achieves a reduction in the complexity of the limited parallelism models.

We provided a formal design of a dialog system using the Z notation. Dialog systems are very much like operating system in the concepts they provide. The specification addresses the invariant properties which need to be satisfied by the various components of the system. In particular, the properties address object relationships in regard to their layout on the graphical interface, presentation of the visual aspects of the objects, activation and execution of programs attached to the objects, and concurrency supported by the system.

K.T.Narayana
Department of Computer Science
Whitmore Laboratory
Pennsylvania State University
University Park, Pa16802
Phone:(814)-863-0147
narayana@shire.cs.psu.edu
Grant Titile: Real-Time System Specification and Verification
Grant Number: N00014-89-J-1171
Reporting Period: Oct 1,1988-Sep 30,1988.

3 Introduction

In recent years the development and the use of real-time systems has increased dramatically. Real-time systems are used in applications where timing behavior is absolutely critical, such as patient monitoring systems, fly-by-wire avionic systems, process control systems, and weapon systems. As such, there is an acute need for the development of formal methods for reasoning about such systems.

Practical limitations often restrict the number of available processors. Processors need not be identical, or even run at the same speed. For example, a system may consist of a single high-speed processor which is used solely for critical tasks, along with one or more slower processors which handle the remainder of the work.

Since the behavior of a real-time system, by its very nature, is highly dependent on the environment in which it is executed, there is a need for semantic models which address such pragmatic concerns. In the paper[ShN88], we provide a linear-history based [FLP84] semantic model for real-time shared variable concurrency under a variety of realistic execution models. We begin with *maximal parallelism* [SaM81], in which every process has its own processor, and then successively generalize the model until we eventually arrive at *asynchronous limited parallelism*, in which the number of processes may exceed the number of available processors and processors may execute at distinct speeds.

We provide a denotational semantics for a small, but realistic, real-time concurrent programming language. The language is based on the notation of Dijkstra'a guarded commands, uses shared variables for process cooperation, and supports atomic actions, process synchronization, and a delay command for real-time control. Features of the language appear in various forms in real-time languages such as Ada [Ada83], CHILL [BLW82], occam [Occ84], and Modula-2 [Wir85]. The primitives of the language are sufficiently low-level that a rich variety of "high-level" constructs can be implemented and analyzed with the framework of the semantic model.

4 Brief Summary of the Semantic Development

We briefly describe the various semantic models (in order of complexity) along with some of the fundamental notions underlying them.

4.1 Machine Models

The formal semantics does not depend on the underlying physical machine used to implement a real-time system. Intuitively, however, the semantics is based on the notion of a MIMD (multiple-instruction, multiple-data) *shared-memory machine*. Such machines consist of one or more physical processors, each of which contains its own local memory. The processors are independent of each other and execute at a fixed speed, although different processors need not execute at the same speed (*i.e.* they may be *asynchronous*). There is also a shared memory

which may be accessed by any processor. We make no assumptions about how conflicting shared-variable accesses are arbitrated.

4.2 The Maximal Parallelism Model

This is the first step in the semantic development. It assumes that processors are synchronized with respect to a discrete global clock. In other words, all processors execute at the same conceptual speed. A unit action takes unit global time on any processor. The fundamental characteristic of this model is that it seeks to locally maximize concurrent activity at every time instant. Specifically,

- Each process is allocated a separate processor on which it is executed.
- Processors do not idle unless prohibited by synchronization or mutual exclusion.
- All shared variables are assumed to be located in a shared memory. Typical memory access models are:
  1. Bus-arbitration: no two processes may access the memory at the same time.
  2. Exclusive-read, exclusive-write: two processes may access the shared memory at the same time only if they are accessing distinct locations.
  3. Concurrent-read, exclusive-write: two processes may read the same memory location simultaneously; otherwise same as above.

  In all of these models, contention is resolved by interleaving. Further, once a process requests access to a shared variable, it is granted access in the least possible amount of time (subject to contention).
- A process executing an atomic action will be granted entry into the action at the earliest time instant the action becomes available.

The maximal parallelism model has somewhat limited practical applicability since it essentially assumes unlimited processor resources. However, it provides an excellent starting point for the formal development and is subsequently generalized.

4.3 Asynchronous Processors

Next, we relax the requirement that the processor speeds be identical; we permit the processors to execute at (possibly) distinct conceptual speeds. While we previously characterized a unit action to take global unit time, we now assume that a unit action takes $\theta(\ell) \geq 0$ global units of time on processor ℓ (equivalently, process ℓ under maximal parallelism). The lower $\theta(\ell)$ is, the faster the processor ℓ. Intuitively, each processor has its own local conceptual clock.

4.4 The Limited Parallelism Model

The maximal parallelism model is relaxed to permit fewer processors than there are processes in the program. The precise number of available processors must be known. Computations are interleaved when necessary, but concurrency is still locally maximized, *i.e.* given a choice, the system must always elect to perform an action rather than idle. We once again assume that all processors (and hence all processes) have identical conceptual speeds.

The development hinges on the notion of an ***instantaneous scheduler***, one which can make and carry out scheduling decisions in zero time; there is no overhead due to context-switching, communication with the scheduler or other implementation dependent details. The scheduler is assumed to have its own dedicated processor. This characterization has nice formal properties and can easily be parameterized to handle more sophisticated scheduling disciplines. Realistic

elements such as context switching delays can be brought to bear as a matter of detail; that is the nice aspect of it. This semantic model is clearly *weaker* than the maximal parallelism model.

It is important to note that limited parallelism and *interleaving* are very distinct models. Interleaving models represent the minimal assumptions necessary to ensure (qualitatively) correct execution of concurrent programs. They impose the least possible implementation restrictions. Limited parallelism, on the other hand, implicitly refers to time by enforcing the real-time criterion of locally maximizing concurrent activity, and makes the (limited) processor resources explicit.

4.5 Asynchronous Limited Parallelism

Unlike asynchronous maximal parallelism, in which a process (and hence its speed) may be identified with that of the processor on which it is executed, here it is useful to distinguish the two concepts. We associate conceptual speeds with processors, and assume that each process can execute on any processor. That is, the set of possible speeds associated with a process (at a given time instant) corresponds to the set of speeds of the available processors (at that instant). The semantic treatment is akin to that considered earlier, with the added complexity that the duration of a unit action in a process may change dynamically as the computation evolves.

4.6 Scheduling Policies

Finally, we address how one can characterize specific scheduling policies in the context of limited parallelism. In particular, we characterize simple priority based scheduling of processes, and nondeterministic time-slicing.

4.7 Full Abstraction

The semantics as formulated is not fully abstract. However under an appropriate notion of observational equivalence of processes, it can be made fully abstract by taking ready closures on the lines of [HGR87]. Unfortunately under limited parallelism, the notion of observational equivalence of processes is extremely sensitive to the number of processors. We need to develop a justifiable basis for observing processes under limited parallelism. When such a characterization is done, formalizing a fully abstract semantics seems useful. We are currently exploring this issue.

5 Language Concepts

The limited parallelism model formalized earlier, locally maximizes concurrent activity, but there need not be a physical processor available for every process. The exact number of available processors is a parameter of the model. Due to the (potentially) limited processor resources it becomes necessary to introduce the notion of a process *scheduler*. As a result, the ability to prove strong quantitative properties of a system is highly dependent on the nature of the scheduler. This increases the complexity of the model. Further, when subcomputations of processes are independent, the ability to attain quantitative bounds under a nondeterministic scheduler is lost. Thus language concepts like processor holding becomes necessary. We develop *fundamental* language concepts which can be used to solve the problems arising in the development of real-time systems under limited parallelism. Consider the following general programming problems:

- Once activated, a device produces an output every t time units until it is deactivated. A process must read the output from the device, evaluate it, output the result of its computations and then wait for a new input. This all must occur within t time units or data will be lost.

- Two or more processes must cooperate in real-time. For example, a motion-detection process and a 3-D recognition process must cooperate to track a moving figure.

Time-dependent problems such as these are commonly encountered in practice. Moreover, they may comprise only a small part of an otherwise time-independent system. In the context of limited parallelism, most real-time programming languages are inadequate for solving such problems; frequently they are just concurrent programming languages augmented with one or two time-dependent constructs such as a **delay** command. In practice, implementations also make use of a real-time *kernel* which provides time-dependent system functions, but from a proof-theoretic standpoint, such extensions are all but worthless.

We introduce in [NaS89] the *tri-section*, a language concept for real-time programming which provides direct solutions for such problems. Tri-sections essentially combine real-time nondeterministic multi-way synchronization with the notion of *processor holding*, in which the execution of a statement may not be preempted by the process scheduler. The concept allows the creation of regions of maximal parallelism in an otherwise limited parallel execution of a program. This makes it possible to infer strong temporal properties within the body of a tri-section, *independent* of the nature of the process scheduler. In practice, this effectively decreases the complexity of the limited parallelism model.

The utility of tri-sections is demonstrated by providing solutions for the following generic problems:

1. A process control system: a simple program which introduces the reader to programming with tri-sections.

2. The *dining philosophers* problem and its generalizations: each of these can be conveniently represented by a class of graphs in which the nodes represent clients (philosophers) and the edges represent resources. In the dining philosophers problem, the graph is a cycle. In the *drinking philosophers* problem it is an arbitrary undirected graph [ChM84], and in the *cooking philosophers* problem it is an arbitrary (undirected) hypergraph.

   The dining and drinking philosophers problems have been explored in the literature [Dij72, ChM84]. However, our solutions are novel because the *resources* ensure that they are properly used, not the clients. Our solution to the drinking philosophers problem differs from Chandy and Misra's in that we do not use auxiliary resources, and require only that the process scheduler be *just* to ensure starvation freedom. Their solution requires strong fairness. To our knowledge, no solution to the cooking philosophers problem has been presented.

3. An *elevator system*: we consider a multi-lift system servicing a building. The elevator system is typical of the synchronization problems encountered in circuit switching systems and channel allocation schemes. Unlike the philosophers problems, here we have a client-server problem. Clients do not know who the servers are. Each client must be matched to exactly one server. A server must be prepared to match with any potential client, and must be able to infer the identity of the client with which it is matched. This is a non-trivial problem when we note that tri-sections do not provide an explicit means of establishing the identity of the partners in a semantic match. We prove conditional starvation freedom of the system with only a *justice* requirement on the scheduler.

Relation to Other Work

In TCSP [Hoa85], Hoare introduces an operator for parallel composition which provides *lock-step* synchronization between processes. However, the participants in a synchronization event are statically determined. CCS [Mil80] permits synchronization between exactly two processes, but synchronization is not mandatory; processes may refuse to synchronize even if given the opportunity to do so. In SCCS [Mil83] it is possible to model multi-way synchronization with

precise timing. However, using the δ operator to model *waiting* for a synchronization event admits the possibility of divergence even if all of the participants are able to engage in the event. Francez [FHT84] introduces multi-way synchronization in a communication abstraction construct known as a *Script*. However, critical roles in Scripts are static; we allow the dynamic specification of the synchronizing set of processes. Further, Scripts (in its present form) does not allow processes to enroll in different scripts simultaneously, *i.e.* it does not permit external nondeterminism; our mechanisms do permit external nondeterminism.

5.1 Tri-Sections

The general form of a tri-section[1] is $\lhd^{\ell}_{X} S \rhd$, where ℓ is the *label*, X is an *index set* consisting of zero or more *indices*, and S is an executable statement. An index is a non-empty set of tri-section labels. X may be a variable or an expression, in which case the underlying language must support the *set* abstract data type. We use the following abbreviations: if X is a constant $\{x\}$, then the outermost braces are dropped, and if X is the constant $\{\{\ell\}\}$, then the tri-section is written $\lhd S \rhd$. Syntactically, tri-sections within different processes must have different labels, although two or more tri-sections with the same label may appear within a process. Semantically, the following restrictions are imposed: 1) $X \neq \emptyset$, 2) for all $x \in X$, $\ell \in x$, and 3) for all $x \in X$, no two of the tri-sections *named* by x (*i.e.* no two labels in x) are contained in the same process.

A set $\{\lhd^{\ell_i}_{X_i} S_i \rhd\}$ of tri-sections is *matching* iff $\bigcap_i X_i \neq \emptyset$. The set is *maximally matching* if $\bigcup_i \{\ell_i\} \in \bigcap_i X_i$. The set is *deadlocked* if it is not matching and for all $x \in X_i$, $(x \setminus \{\ell_i\}) \cap \bigcup \{\ell_i\} \neq \emptyset$.

The operational semantics of a tri-section is as follows. When a process begins execution of $\lhd^{\ell_0}_{X_0} S_0 \rhd$, the tri-section ℓ_0 is said to be *enabled*. The process is suspended immediately before execution of S_0, and waits until there exists a maximally matching set $M = \{\lhd^{\ell_i}_{X_i} S_i \rhd\}$ of enabled tri-sections containing ℓ_0. The *first* such M (in real time) is always chosen. If two or more maximally matching sets are established simultaneously, the *oldest* match is chosen, *i.e.* the one which contains a tri-section which has been enabled the longest; if two maximal matches are the same *age*, one is chosen nondeterministically. Each suspended process[2] in M is then granted an *available* physical processor as soon as possible. A processor is *available* if it is not executing a tri-section (in which case the process running on it may be preempted). If $|M|$ is greater than the number of physical processors in the system, then the program *aborts*. The physical number of processors is a parameter of the model. Once all of the processors are acquired, M is *executed*, *i.e.* the tri-sections in M are initiated for execution *simultaneously*. Thus, for all $\ell_j, \ell_k \in M$, tri-sections ℓ_j and ℓ_k will begin execution of S_j and S_k concurrently. Each process ℓ_i *holds* its processor (may not be preempted or re-scheduled) until it has finished executing statement S_i. This ensures that concurrent activity is locally maximized throughout the execution of the tri-section.

Tri-sections combine two real-time concepts: processor holding and nondeterministic multi-way synchronization. Consider the tri-section $\lhd S \rhd$. The operational semantics says that the processor on which the tri-section is executed should be *held* until the execution of S is completed. In effect, this introduces a syntactically determined region of maximal parallelism into the program. As such, strong temporal properties of S can be established, since the execution of S is not subject to interleaving by the process scheduler. One might be inclined to regard $\lhd S \rhd$ as the real-time analogue of the *atomic action* $\langle S \rangle$. However, the resemblance is not so strong as it might seem. There are at least three fundamental differences:

1. Atomic actions *must* terminate [ScA85]; tri-sections need not.

2. In order to ensure termination, there may be an arbitrary delay before an atomic action is executed. However, a tri-section $\lhd S \rhd$ will be *executed* after a bounded implementation-

[1] The notation for tri-sections was developed before the name; the term *tri-section* is derived from the shape of the symbols "$\lhd$" and "$\rhd$".

[2] We will sometimes refer to a tri-section and the process in which it appears interchangeably.

dependent delay, since there is an available processor (namely the one on which the tri-section is running).

3. Atomic actions are *indivisible*; the intermediate states of S are not visible to the environment. The intermediate states of a tri-section are completely visible to the environment (unless they are hidden by some other means), and tri-sections impose no restrictions on concurrent behavior (other than those implicit in the physical processor requirements).

The synchronization aspect of a tri-section is necessary for those situations in which two or more processes must cooperate in real-time. It is well-known that time-independent synchronization between n processes (which essentially means ensuring that each process has reached a particular state) can be programmed in an interleaving environment with $O(n)$ synchronous interprocess communications. Thus it might seem that tri-sections need not include synchronization, since it can be programmed by other means. However, this is *not* the case, for the following reasons:

1. Tri-sections can exhibit *external nondeterminism*. When a process executes $\lhd^{\ell}_{X} S \rhd$, where $|X| > 1$, the choice of who the process will synchronize with is made by the *environment*, not the process itself. Further, the process is unaware of which choice was made.

2. The synchronization part of a tri-section is established *in real time*; the first maximally matching set of tri-sections is chosen.

3. Tri-sections are *literally* synchronized. Given a maximally matching set $\{\lhd^{\ell_i}_{X_i} S_i \rhd\}$ of enabled tri-sections, the S_i are initiated *simultaneously*.

The synchronization part of a tri-section is in fact extremely useful in its own right, independent of processor holding. As a result, we use the notation Δ^{ℓ}_{X} as an abbreviation for the tri-section $\lhd^{\ell}_{X}$`null` $\rhd$, where `null` is the empty statement.

Weak Tri-sections

Tri-sections require a process to name *all* of the tri-sections with which synchronization should occur. Occasionally this requirement is too strict; a process may only know with whom *it* would like to synchronize, and may not care if additional processes are involved. Thus we introduce the *weak tri-section* $\trianglelefteq^{\ell}_{X} S \trianglerighteq$, which is semantically equivalent to $\lhd^{\ell}_{Y} S \rhd$, where

$$Y = \{y : \exists x \in X (y \text{ is a valid index set} \wedge x \subseteq y)\}.$$

This definition is adopted for convenience; it is not realizable as written, since the set Y may be infinite. However, a practical, efficient implementation can be given. As above, we will take $\underline{\Delta}^{\ell}_{X}$ to be an abbreviation for $\trianglelefteq^{\ell}_{X}$`null` $\trianglerighteq$. We use the term *strong tri-section* to refer to a tri-section which is not weak.

For simplicity, we insist that any set of maximally matching tri-sections contain at least one *strong* tri-section. The other definitions remain as before.

Proof Obligations

Tri-sections are inherently *unfair*. Most process schedulers are at least *weakly fair* (or *just*). That is, if an action of a process is continuously enabled from some point onward, then eventually that action will be executed. If the number of processes in a system exceeds the number of physical processors, it is possible that all of the processors may be utilized by non-terminating tri-sections, preventing other enabled processes from executing. Thus, in order for an otherwise fair scheduler to be fair in the presence of tri-sections, it is necessary to prove that this situation cannot arise. Two practical strategies for doing so are either to prove that all tri-sections

terminate, or prove that at least one processor is always available (*i.e.* that the cardinality of a maximal match is always strictly less than the number of available processors).

A set of enabled tri-sections which is *deadlocked* is clearly unable to make progress and will wait forever for execution. Since the semantics admits the possibility of deadlock, the programmer must prove that such a situation cannot arise.

6 Formal Specification of Dialog Systems

Dialog systems are servers for an interface; graphical interfaces are one such. They are like operating systems in the concepts they provide. From a functional point of view, they maintain the interface for the application, permit concurrent execution of programs attached to graphical objects on the interface, and provide services with which a user (or programs) can edit the object of the interface. We formulate the invariant properties which need to be satisfied by the various components of the system. These properties involve treatment of object relationships in regard to their layout, activation and execution of programs attached to objects, concurrency model supported by the system, and the presentation of visual aspects. A formal specification of the system has been developed in [Nar89,NaS89,NaS89]. This specification addresses the various aspects in the design of dialog systems.

References

[Ada83] *Reference Manual for the Ada Programming Language*, United States Department of Defense, Washington, 1983.

[BLW82] P. Branquart, G. Louis and P. Wodon, "An Analytical Description of CHILL, the CCITT High-Level Language VI," *LNCS* **128**, 1982.

[ChM84] K.M. Chandy, J.Misra, "The Drinking Philosophers Problem," *ACM TOPLAS*, 6, 4, Oct 1984, pp. 632-646.

[Dij68] E.W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages*, Academic Press, New York, 1968.

[Dij72] E.W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *in Operating System Techniques*, Ed. Hoare and Perrott, Academic Press, 1972.

[FHT84] N. Francez, B. Hailpern, G. Taubenfeld, "Script: a Communication Abstraction Mechanism and Its Verification" in *Logics and Models of Concurrent Systems*, K.R. Apt (ed), 1984, pp 169–212.

[FLP84] N. Francez, D. Lehman and A. Pneuli, "A Linear-History Semantics for Languages for Distributed Programming," *TCS* **32**, 1984, pp. 25–46.

[Hoa85] C.A.R. Hoare, "Communicating Sequential Processes," *Prentice-Hall International* 1985.

[HGR87] C. Huizing, R. Gerth and W.P. de Roever, "Full Abstraction of a Real-Time Denotational Semantics for an OCCAM-like Language," in *Proc. 14th ACM Symposium on POPL*, 1987, pp. 223–238.

[Lam85] L. Lamport, "On Interprocess Communication," *DEC SRC Report*, 8, 1985.

[Mil83] R. Milner, "Calculi for Synchrony and Asynchrony," *Theoretical Computer Science*, 25, 1983, pp. 267-310.

[Mil80] R. Milner, "A Calculus of Communicating Systems," *LNCS* **92**, Springer-Verlag, 1980.

[Nar89] Narayana,K.T, *A Formal Model and A Specification of a Dialog System*, Technical Report, Department of Computer Science, Pennsylvania State University, University Park, Pa16802.

[NaS89] K.T. Narayana and E.Shade, "Language Concepts for Real-Time Concurrency," *Research Report*, Department of Computer Science, The Pennsylvania State University, University Park, Pa16802.

[NaS89] Narayana,K.T and Sanjeev Dharap, *Formal Specification of a Look Manager*, Technical Report, Department of Computer Science, Pennsylvania State University, University Park, Pa16802.

[NaS89] Narayana,K.T and Sanjeev Dharap, *Invariant Properties in a Dialog System*, Technical Report, Department of Computer Science, Pennsylvania State University, University Park, Pa16802.

[Occ84] *The OCCAM Language Reference Manual*, Prentice-Hall, London, 1984.

[SaM81] A. Salwicki and T. Müldner, "On the Algorithmic Properties of Concurrent Programs," *LNCS* **125**, Springer-Verlag, 1981.

[ScA85] F.B. Schneider and G.R. Andrews, "Concepts for Concurrent Programming," *LNCS* **224**, Springer-Verlag, 1985, pp. 669-716.

[ShN88] E.Shade and K.T. Narayana, "Real-Time Semantics for Shared-variable Concurrency," *Computer Science Research Report*, July 1988.

[Wir85] N. Wirth, *Programming in Modula-2*, 3/e, Springer-Verlag, Berlin, 1985.

K.T.Narayana
Department of Computer Science
Whitmore Laboratory
Pennsylvania State University
University Park, Pa16802
Phone:(814)-863-0147
narayana@shire.cs.psu.edu
Grant Titile: Real-Time System Specification and Verification
Grant Number: N00014-89-J-1171
Reporting Period: Oct 1,1988-Sep 30,1988.

Published Papers

K.T.Narayana, A.A. Aaby, Specification of Real-Time Systems in Real-Time Temporal Interval Logic, IEEE Symposium on Real-Time Systems, Dec 1988.

Papers Submitted for Publication

Real-Time Semantics for Shared Variable Concurrency (with Eric Shade)
Language Concepts for Real-Time Concurrency (with Eric Shade)
Formal Specification of a Look Manager (with Sanjeev Dharap)
Invariant Properties in a Dialog System (with Sanjeev Dharap)

Technical Reports

Real-Time Semantics for Shared Variable Concurrency (with Eric Shade)
Language Concepts for Real-Time Concurrency (with Eric Shade)
Formal Specification of a Look Manager (with Sanjeev Dharap)
Invariant Properties in a Dialog System (with Sanjeev Dharap)
Synthesis of Hardware Elements from Propositional Temporal Interval Logic (with A.A. Aaby)
A Formal Model and a Specification of a Dialog System

THE PENNSYLVANIA STATE UNIVERSITY
AUXILIARY ACCOUNTING SYSTEM

DATE: 10/09/89; TIME: 12:14
PAGE: 1
PROGRAM: PG0467D

REPORT NAME: AUXILIARY BUDGET REPORT
REQUESTOR: SCIENCE

FISCAL YEAR: 1 BUDGET: 428-28 COMPUTER SCI FUND: 72620 DN VERIFICATION ADMIN OFFICER: GLG

PROJECT: DIRECTOR: DR. K. NARAYANA INITIATION DATE: 10/01/88 TERMINATION DATE: 09/30/89 GRANT #: N00014-89-J-1171
COST SHARING RATE: 0.00 FRINGE BENEFIT RATE S/W: 26.40/ 8.10 INDIRECT COST CODE: B INDIRECT COST RATE: 44.40

REPORT CONTROL SPECIFIED BY REQUESTOR: 7

DESCRIPTION	PERMANENT BUDGET	TEMPORARY BUDGET	TOTAL BUDGET	ENCUMBRANCES	ACTUALS	BALANCE
SALARIES AND WAGES						
SALARIES						
FACULTY SALARIES	14,401.00	0.00	14,401.00	.00	14,145.20	255.80
FI -ACADEMIC	0.00	0.00	0.00	.00	.00	.00
FII-ACADEMIC	0.00	0.00	0.00	.00	.00	.00
VISITING - ACADEMIC	0.00	0.00	0.00	.00	.00	.00
GRAD ASST	11,501.00	0.00	11,501.00	.00	7,100.00	4,401.00
STAFF AND TECH SVC	939.00	0.00	939.00	.00	.00	939.00
TOTAL SALARIES	26,841.00	0.00	26,841.00	.00	21,245.20	5,595.80
WAGES	0.00	0.00	0.00	.00	.00	.00
TOTAL SALARIES AND WAGES	26,841.00	0.00	26,841.00	.00	21,245.20	5,595.80
OTHER EXPENSE						
SUPPLIES	150.00	0.00	150.00	.00	.00	150.00
COMMUNICATION SVCS	250.00	0.00	250.00	.00	8.50	241.50
DOMESTIC	2,000.00	0.00	2,000.00	.00	967.40	1,032.60
FOREIGN	0.00	0.00	0.00	.00	.00	.00
PUBLICATIONS	500.00	0.00	500.00	.00	.00	500.00
MISCELLANEOUS	2,250.00	1,457.20-	792.80	.00	.00	792.80
TUITION AND FEES	3,842.00	0.00	3,842.00	.00	2,280.00	1,562.00
TOTAL OTHER EXPENSE	8,992.00	1,457.20-	7,534.80	.00	3,255.90	4,278.90
FRINGE BENEFITS	4,838.00	76.56-	4,761.44	.00	4,044.34	717.10
EQUIPMENT	2,000.00	0.00	2,000.00	.00	.00	2,000.00
OVERHEAD	12,423.00	1,533.76	13,956.76	.00	9,614.00	4,342.76
GRAND TOTAL ACTUAL EXPENSE	55,094.00	0.00	55,094.00	.00	38,159.44	16,934.56

END OF REPORT