CONCEPTUAL FRAMEWORKS FOR DISCRETE EVENT SIMULATION MODELING

by

Emory Joseph Derrick

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

MASTER OF SCIENCE

in

Computer Science and Applications

APPROVED:

Osman Balci

Osman Balci, Chairman

Richard E. Nance

James D. Arthur

Jeffrey D. Tew

August, 1988

Blacksburg, Virginia

-2

CONCEPTUAL FRAMEWORKS FOR DISCRETE EVENT SIMULATION MODELING

by

Emory Joseph Derrick

Committee Chairman: Osman Balci
Computer Science

(ABSTRACT)

This thesis examines those aspects of simulation with digital computers which concern the use of conceptual frameworks (CFs) for the design and implementation of a model. A literature review of CFs which are in common use is conducted. These CFs are applied to a complex modeling problem, a traffic intersection system. A comparative review of the CFs is given based upon the lessons learned from the above applications, and a taxonomy is developed.

The research clarifies the differences that exist among the myriad of CFs in use today. In particular, the comparative review highlights the significant CF features that are necessary for successful model representation of discrete-event systems. The taxonomy provides a useful and meaningful classification of CFs and produces insights into the conceptual relationships that exist among them. The characteristics of CFs that are desired to enable the development of model specifications that are analyzable, domain independent, and fully translatable are identified. The roles of CFs are better understood and specific potential directions for future research are pinpointed.

ACKNOWLEDGEMENTS

First, I deeply thank my wife, Ruth, for her unselfish support of this work. My three children have also contributed significantly with the sacrifice of time with their father.

I owe special thanks to Dr. Osman Balci for his enthusiasm and technical guidance. Also, Dr. Richard Nance has provided encouragement and his time to discuss the content and ideas that are presented. With much appreciation, I thank Dr. Robert Moose who devoted a great deal of time and energy to help me understand the fine art of using job control and text formatting languages.

Last, I gratefully acknowledge the support of this research by the United States Navy through the Systems Research Center of Virginia Tech.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ACD	—	Activity Cycle Diagram
ACOS	—	ASP Common Operational Software Support
AS	—	Activity Scanning
ASP	—	Advanced Signal Processor
BERM	—	Binary Entity-Relationship Model
CAP	—	Condition Action Pair
CF	—	Conceptual Framework
CM	—	Conical Methodology
COL	—	Current Objects List
CS	—	Condition Specification
CSL	—	Control and Simulation Language
DEVS	—	Discrete EVent System Specification
EAS	—	Entity-Attribute-Set
ECOS	—	EMSP Common Operational Software Support
ECSL	—	Extended Control and Simulation Language
EMSP	—	Enhanced Modular Signal Processor
ER	—	Entity-Relationship
ERA	—	Entity-Relationship Approach
ES	—	Event Scheduling
FIFO	—	First-In, First-Out
FOL	—	Future Objects List
GASP	—	General Activity Simulation Program
GC	—	Graph Control
GERM	—	Generalized Entity-Relationship Model
GIP	—	Graph Instantiation Parameter
GPSS	—	General Purpose Simulation System
GSP	—	General Simulation Program
GV	—	Graph Variable
HOCUS	—	Hand Or Computer Universal Simulator
IC	—	Integrated Circuit
MG	—	Model Generator
MMS	—	Model Management System

NEP — Node Execution Parameter
OOP — Object-Oriented Paradigm
PGM — Process Graph Method
PI — Process Interaction
PIP — Primitive Interface Procedure
SM — Structured Modeling
SMDE — Simulation Model Development Environment
SML — Structured Modeling Language
SMSDL — Simulation Model Specification and Documentation Language
SPGN — Signal Processing Graph Notation
SPL — Simulation Programming Language
STA — System Theoretic Approach
TF — Transaction Flow
TI — Traffic Intersection
TPA — Three-Phase Approach

CHAPTER 1

INTRODUCTION

Simulation studies are assuming an increasingly important role in our growing technological society. Experts [Shannon 1975; Emshoff and Sisson 1970; Fishman 1973] agree that when simulation is appropriate for a given problem, significant advantages are available to the modeler in his quest for meaningful problem solutions. Shannon [1975] defines **simulation** as

> "the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behavior of the system or of evaluating various strategies (within the limits imposed by a criterion or set of criteria) for the operation of the system."

Simulation may be undertaken using various computational tools, most notably analog, digital, or hybrid computers [Balci 1986] and it may be applied to problem domains which are suitably solved by four known techniques. These techniques, described below, are :

- Monte Carlo methods — A "static, distribution sampling kind of simulation" which is "traditionally used to estimate probabilities of a model's states through sample-driven experimentation" [Kreutzer 1986].
- continuous — Simulations of systems in which a model's states change continuously with time, represented by differential and/or difference equations [Balci 1986].
- discrete-event — Simulations of discrete systems where model state changes occur only at discrete, fixed points in time.

- combined — Simulations of systems with both continuous and discrete-event components.

The ability of a modeler to accurately accomplish the design of the model is one of the critical concerns which face those who employ or will employ simulation for the determination of problem solutions. The importance of the model formulation and representation processes of a simulation study's life cycle [Balci 1986] must not be underestimated. The accomplishment of these processes which produce the conceptual and communicative forms of the model takes place under the influence of a *conceptual framework* [Balci and Nance 1987b]. We define a conceptual framework (CF) to be:

> an underlying *structure and organization of ideas* which are the *outline and basic frame* that **guide** the modeler in representing a system in the form of a model.

The research described by this thesis focuses on those aspects of simulation with digital computers which concern the use of CFs for the design and implementation of the model (or representation of the system of interest). Furthermore, since the research directly supports the SMDE (Simulation Model Development Environment) [Balci and Nance 1987a, 1987b], we limit our concerns to the use of CFs as applied to discrete-event systems only.

1.1 The SMDE and the Importance of CF Research

Balci and Nance [1987a, 1987b] describe the ongoing research at Virginia Tech to develop a prototype SMDE which aims "to provide an integrated and comprehensive collection of computer-based tools" for automated support in model development of discrete-event systems. An overview of the architecture of the SMDE is shown in Figure 1.1. Such an automated environment will offer substantial, cost-effective gains for

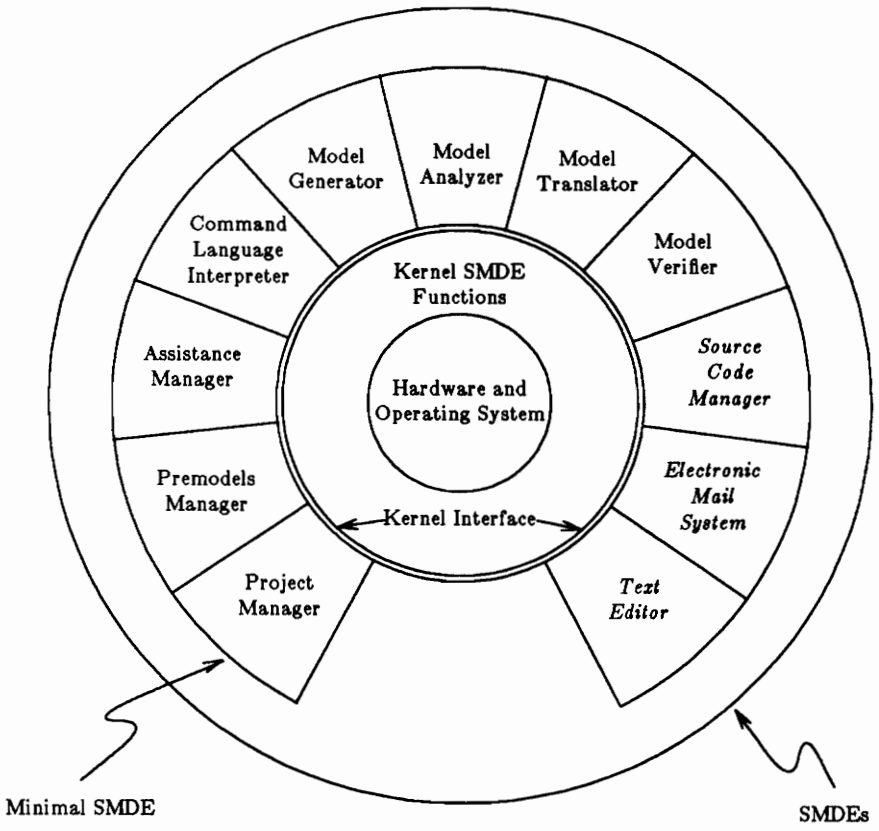


Figure 1.1 The Architecture of the SMDE Research Prototype [Balci and Nance 1987b]

simulation studies in model quality, project team efficiency and productivity, and in reductions to model development time [Balci and Nance 1987b]. The Model Generator (MG) tool, one of several tools available to the modeler within the SMDE, is that tool which assists the modeler in the performance of the model formulation and representation processes discussed above. The MG tool converts the conceptual model into a formal specification while under the influence of a CF [Balci and Nance 1987b].

Herein lies the importance of the CF research. The SMDE project goal is to achieve the automation-based paradigm [Balzer et al. 1983] within the SMDE, via the evolutionary development of several prototypes [Balci and Nance 1987b]. The specification which is created by the modeler under the influence of a CF must be fully translatable into executable code for the automation-based paradigm to be realized. In addition, it is essential that the specification be formally analyzable and domain independent [Balci and Nance 1987b]. Because of the heavy reliance of the MG tool upon a CF, it is vital that a CF or CFs be utilized that will produce these desired features in the resulting specification. Research and study are needed to support CF selection.

1.2 Description of Research

An appropriate CF or CFs upon which to base the MG tool is required. This research explores the realm of CFs in order to support this requirement. In particular, we seek to gain an understanding of CFs through the study of the current literature, through the practical experience of applying CFs to a complex study problem, and through the accomplishment of a comparative review and a taxonomy formulation. Our goal is to make measurable headway in understanding those features of a CF which support the development of specifications which are analyzable, domain independent, and

more importantly, completely translatable.

Chapter 2 describes the literature review of CFs which are in common use today. Applications of the CFs to a traffic intersection system are described in detail in Chapter 3. This work produces insights into the capabilities and limitations of the CFs. In some cases, these applications represent a first-time accomplishment in demonstrating model representation of a complex system. The comparative review of the CFs, given in Chapter 4, in hand with the experience derived from the applications of Chapter 3, enable the development of the taxonomy of CFs in Chapter 5. Chapter 6 summarizes those features which we feel are essential for the CF or CFs which will support the SMDE within the MG tool. Furthermore, Chapter 6 offers conclusions and potential areas for future research.

CHAPTER 2

LITERATURE REVIEW

The literature has been extensively reviewed to gain a grasp on the conceptual frameworks currently being used in model and system design efforts which show promise in stimulating improvements to discrete event model representation techniques. The results of this review form the basis for the later sections. First, the terminology of the discrete event modeling domain is clarified. Due to the wide range of terminology and interpretations by simulation experts, the importance of a sound definitional base to introduce such a review cannot be overestimated [Nance 1981b]. A discussion of time flow mechanisms is next covered and is essential for a clear understanding of the conceptual frameworks which are low-level in nature [See Sections 2.3 through 2.7]. Finally, each conceptual framework of interest which has been identified is described in a tutorial fashion.

2.1 Time and State Definitions

Nance [1981b] recognized the need for an "integrating general framework" for approaches to model development in discrete event simulation. He states that the "independent and concurrent development of several SPLs [simulation programming languages] during 1960-1963 and shortly thereafter" occurred in a progressive environment suffering from the lack of a fundamental theoretical basis. One problem which resulted from these conditions was the infiltration into the simulation literature of wide and subtle variations in the definitions of terms and concepts which are basic to simulation model development. Differences arose surrounding the terms *event*, *activity*, and

process which are at the very core of understanding time and state relationships. Nance [1981b] offers a set of basic definitions which seek to resolve this problem. In the definitions that follow, a system model is made up of *objects* and the *relationships* that exist between them. The concept of the model object, "anything that can be characterized by one or more *attributes* to which *values* can be assigned" [Nance 1981b], is used as the "link" to resolve the definitional differences. The terms *object* and *entity* are regarded as synonomous since both refer to a model component; the term *object* will be used hereafter.

Definitions [Nance 1981b] that revolve around the concept of system time, a common "indexing" attribute among simulation models, include:

- *instant* — "a value of system time at which the value of at least one attribute of an object can be assigned"
- *interval* — "the duration between two successive instants"
- *span* — "the contiguous succession of one or more intervals"
- object *state* — "the enumeration of all attribute values of that object at a particular instant"

Finally, to conclude this section, the definitions for event, activity, and process from Nance [1981b] are given:

- *event* — "a change in object state, occurring at an instant, that initiates an activity precluded prior to that instant."
- *activity* — "the state of an object over an interval"
- *object activity* — "the state of an object between two events describing successive state changes for that object"

- *process* — " the succession of states of an object over a span (or the contiguous succession of one or more object activities)."

2.2 Time Flow Mechanisms

Time flow mechanisms are the methods by which the system clock is updated. In other words, a model's time flow mechanism is the means of time sequencing by which a model progresses in its execution and in its attempt to mimic the system for which it has been built. There are two general categories of time flow mechanisms that are used in discrete event simulation models, the *fixed-time increment* and the *variable-time increment* methods.

The fixed-time increment method (also known as interval-oriented simulation, uniform time increment, or synchronous method [Neelamkavil 1987]) dictates that model time is updated at fixed time increments or steps of constant time. As each time increment passes, model objects are examined to determine what attributes, if any, need to be updated.

The variable-time increment method (also known as event-oriented simulation, next event method, and asynchronous method [Neelamkavil 1987]) however, provides an advantageous means for updating the system clock or global time. The time(s) during which events are not occurring can be skipped. This dead time can then be removed from the model without affecting its execution. The variable-time increment method is the approach which is commonly used in discrete event simulations and can be applied to a wide range of simulation strategies [Neelamkavil 1987].

In addition to the above methods, Nance [1971] provides a stimulating discussion describing the important concepts surrounding time flow mechanisms and modifications

to these approaches in the context of the patrolling repairman problem.

2.3 Event Scheduling (ES)

When using this particular viewpoint in modeling, the modeler considers the system of interest to be composed of events which are determined from a detailed study of the system. Each identifiable event is associated with a series or grouping of actions that contain all the necessary information to at least influence the required state change(s) which are related to that event. Such a grouping can be called an event routine. Pidd [1984] defines an event routine to be "a set of actions that may follow from a state change in the system." Kiviat [1969] describes the approach as one which seeks to execute the event routine "only when a state change occurs." He further suggests that this approach specifies that "some event is to take place at a determined time in the future" which he calls "by predetermined instruction." Therefore, by explicitly scheduling the event routines at a future determined time in accordance with the observable interactions and relationships among system components, system behavior can be represented by the model for any given period of time. The scheduling of event routines is managed during implementation by the maintenance of a list called the "event list" [Pidd 1984]. The event list is a list of event notices or records which are ordered by time.

Figure 2.1 is a simple pictorial flowchart of the basic algorithmic structure of the Event Scheduling Conceptual Framework (ES CF). After initializations, the next event is selected. (The system clock is also updated at the same time.) The event routine which is associated with the next event is then executed. Next, when applicable, the conditions for termination of the simulation are checked. If these conditions are satisfied, the output statistics are then calculated and displayed and the simulation ends. Otherwise, the next

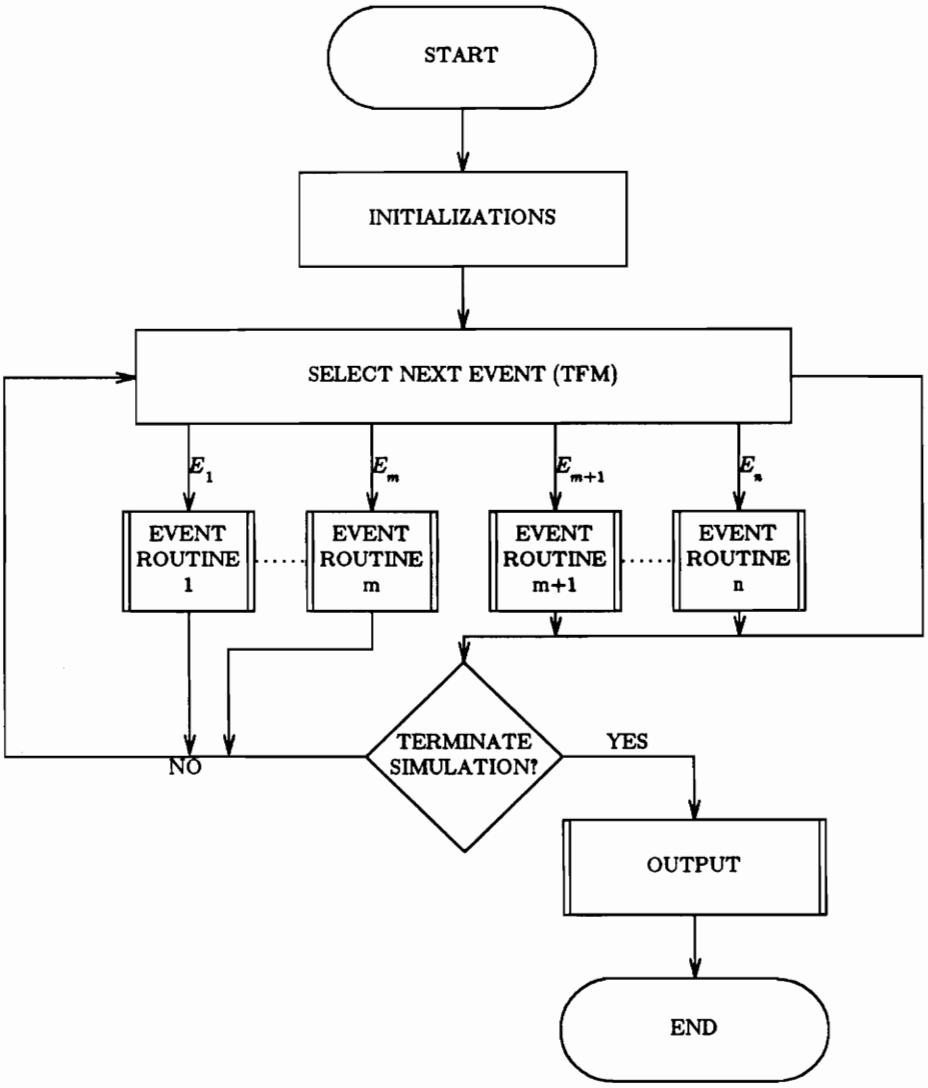


Figure 2.1 The Event Scheduling Conceptual Framework [Balci 1988]

event is selected and the algorithm repeats itself.

When simulation termination conditions are based on the number of observable jobs or departures (server/queueing problems), some event routines (e.g., arrivals) do not affect the terminating conditions. Upon completion of these routines, the terminating conditions do not need to be checked. Other event routines (e.g., departures) have a definite impact on the terminating conditions and a check of these conditions becomes necessary. If time is the basis for the terminating conditions, the execution of a next event may be bypassed when the terminating time occurs before the scheduled time of the next event.

The key elements of the algorithm's structure, the selection of the next event and the event routines and their components, are now discussed in the next two sections.

2.3.1 Selection of Next Event

When discussing the ES CF, it is important to mention how the selection of the next event is made. Since the event records are ordered in some fashion according to time, the next event is simply that event on the event list with the earliest time. In cases where several events are to occur at the same time, precedence rules must be established to break these ties. The modeler is only concerned with the occurrence of events. This is an implementation of the variable-time increment time flow mechanism.

2.3.2 Typical Components of Event Routines

Event routines may contain the following types of actions or items:

- Creation or destruction of event records

The event record most often contains the time at which the event is to occur, an event descriptor or identification label (such as arrival, departure, etc.) which logically links the record to its corresponding routine, and key attributes and values of the model or its associated submodels. Figure 2.2 illustrates an event list of typical records. The number of event records which are maintained on the event list directly affects the implementation efficiency. Therefore, the creation and destruction of these records must be carefully considered.

- Scheduling of future events

The occurrence of an event may dictate the scheduling of some other future or concurrent event by placing it on the event list. The scheduling may be deterministic (event timing determined by trace input) or stochastic (events determined by sampling statistical distributions) [Kreutzer 1986]. In this way, the ES CF provides bootstrapping techniques which enable the generation/regeneration of events. Such techniques perform the explicit scheduling of events and allow the model to produce system-like behavior and to progress toward a successful termination. In other words, since an event occurs at a single instant of time, the scheduling of events accomplishes the "passage of time" in the model [Kreutzer 1986]. For example, in queueing-type models, arrival events often generate a following arrival or a pending departure when the server can be immediately engaged. A departure event similarly generates another departure event when a job, waiting for service, can be assigned to a released server. Fishman's [1973] diagrams very clearly show how the algorithm of this framework is accomplished. The explicit scheduling of events results in a clean and smooth model execution, improving efficiency. Yet, as Kreutzer [1986] points out, the model logic becomes fragmented with the scattering of scheduling commands as the number of event routines and their potential interactions

EVENT OCCURRENCE TIME	*EVENT IDENTIFICATION*	*ATTRIBUTE 2*	*ATTRIBUTE 3*	...	*ATTRIBUTE k*

Figure 2.2 A Typical Events List
[Balci 1988]

increase.

- Contingent events

Events may be contingent or determined [Nance 1981b]. A contingent event occurs at a future time which is unknown and which specifies the time at which some set of boolean conditions becomes true. A determined event occurs at a known future time. When possible, a contingent event is included within the event routine of a determined event [Fishman 1973]. (Fishman [1973] refers to contingent events as *conditional.*) This reduces the number of event records which must be processed and improves model efficiency. With simple models, such as a simple queueing model, conditional events (like change of server status to busy or idle) are less difficult to incorporate into existing unconditional event routines. However, when the conditions upon which the event is to occur become more complex, the difficulty increases dramatically. A good example of this occurs in the traffic intersection problem which is discussed in a later section. The implementation also becomes less readable, less understandable, and harder to debug with the increased complexity of conditions.

2.4 Activity Scanning (AS)

The AS CF was developed in the late 1950's in England and became popular for use in simulation languages like GSP (the General Simulation Program) and CSL (Control and Simulation Language) [O'Keefe 1986b]. The AS CF and an extension of it, the Three-Phase Approach (discussed in a later section), have unfortunately received much less attention and are not well understood in the United States. Much of the literature which describes the AS CF is dated by 20 to 25 years and centers on the original versions of CSL and its forerunners [Kelley and Buxton 1962; Buxton and Laski 1962; Laski 1965;

Buxton 1966; Kiviat 1969]. Beyond Pidd [1984] and Kreutzer [1986], there are other good descriptions among the recent literature [Fishman 1973; Hooper and Reilly 1982; Hooper 1986a, 1986b; Zeigler 1976].

The AS CF requires that the modeler identify the various types of objects in the system to be modeled, the activities which the objects perform, and the conditions under which these activities take place. In particular, the state transition actions that immediately follow a state change for an object must be indicated for each activity. The test set of boolean conditions, or the "testhead" [Pidd 1984] that is associated with these actions, enables the determination of the state change that will initiate that activity. The testheads serve to link the various activities together and to produce the state transitions of the model objects and the interactions among them. In this way, the model is made up of modules or segments of testheads and associated actions which await execution at the appropriate time [Pidd 1984].

It is extremely important to remember that the activity is a state of an object which is bound by successive events of interest. In actual practice, the precise definition of the word "activity" has become somewhat muddled. Traditionally, "activities" are often created which occur in zero simulated time. Examples of this are arrival activities, departure activities, and other functional activities needed to accomplish a specific implementation purpose (such as the SPLIT activity in the AS CF application described in Chapter 3). In addition, activities which occur over some time duration are split into two separate "activites", a beginning activity and an ending activity. For example, a service activity becomes a begin service activity and an end service activity. In this case, the original activity has been transformed (in actuality) into events (not activities) which bound the true activity.

Kreutzer's [1986] discussion of the "activity description" is much less ambiguous than traditional approaches. Kreutzer suggests that the activity description of a "time-consuming activity" retain the "notion of causally connected start and finish events." He further indicates that this could be accomplished by the use of a Pascal-like CASE structure as the body of the activity. Such a structure would have two entry points, one for the "start" event and another for "finish". Each would be prefaced by its testhead as discussed above. Thus, Kreutzer's encapsulation of the bounding events within the single activity description (rather than splitting them apart) is more straightforward.

In general, the "activity descriptions" (whether encompassing the full essence and meaning of "activity" as suggested by Kreutzer or split into separate parts in the more traditional way) form the basis for the AS CF. From the above discussion, one should realize that the use of the term "activity" is widely accepted to refer to the start and finish events within an activity-oriented perspective. Thus, for the remainder of this paper, in discussing activity-oriented conceptual frameworks, the terms "start activity" and "finish or end activity" will be used. In describing activities of zero duration, others have adopted the term "event" which is more precise [Davies and O'Keefe 1987] but may lead to further confusion when used within the context of activity-oriented conceptual frameworks.

Implementations of the AS CF include a monitor or executive which performs a time scan to ascertain the time increment or update to the system clock. Following the time scan, the monitor then conducts an activity scan for the current timing cycle. The activity scan is a check of all testheads to determine which of the activities are to be next executed [Pidd 1984]. Figure 2.3, a flowchart of the monitor's algorithm, clearly shows this two-phase structure (time scan and activity scan). These two scans are discussed in

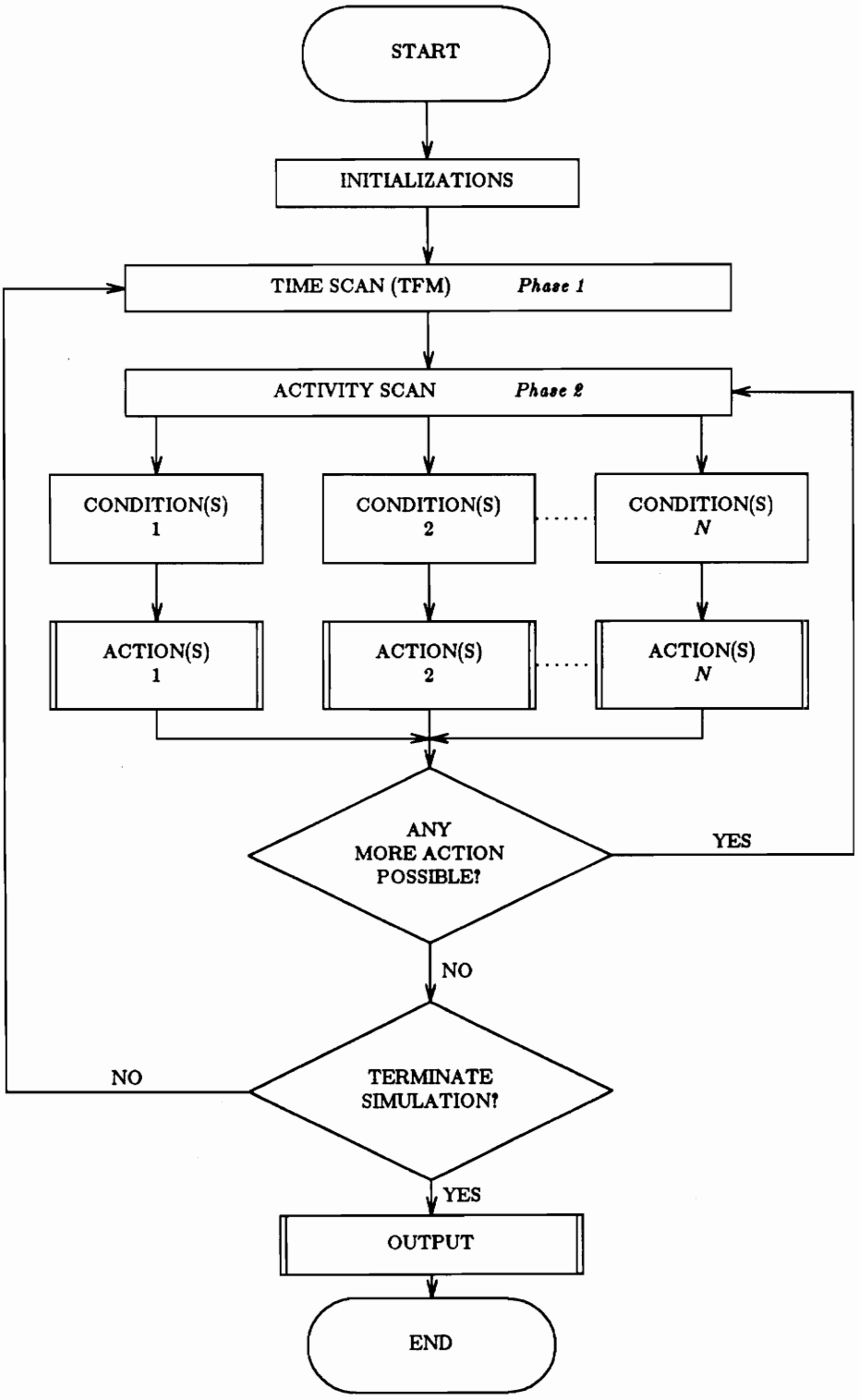


Figure 2.3 The Activity Scanning Conceptual Framework [Balci 1988]

more depth in the next two sections.

2.4.1 The Time Scan

The selection of the next event time is commonly accomplished by maintaining a local clock with each object that changes state. For a simple single server queueing problem, this would mean maintaining such a clock for the next arriving object and also for the server object [Fishman 1973]. Pidd [1984] calls these local clocks "time cells" (or t-cells) and identifies them as attributes of permanent objects which indicate when each object is to change state. Certain non-existent pseudo-objects often need to be created which will hold the necessary time-cell values. The time cells can be maintained in a list (ordered or unordered). The next event time is the minimum time cell value (which is greater than current system time) among the time cells on the list. Time cell values which are less than the current system time indicate that the associated object is idle or waiting to be engaged or committed. Pidd further suggests that the time cell values can be recorded in absolute system time or as a time interval to the state change. Laski [1965] and Buxton [1966] provide excellent coverage on the use of time cells.

An alternate method of accomplishing the time scan (rather than attach t-cells to objects) is to attach the t-cell to the activity itself [Kreutzer 1986]. (Although Kreutzer's discussion is strictly oriented to the Three-Phase Approach, the idea is nevertheless applicable to the pure AS CF.) In this case, the t-cell is (in some way) attached to end or finish activities. The update of time is accomplished as discussed above. A close examination reveals that these methods are, in fact, equivalent. Object t-cells hold time values which are associated with finish activities.

An important distinction should now be clarified. The ES CF associates the next event time with the next event through the use of an events list. Yet, the AS CF transitions to the next event through the logical checks of testheads as discussed above. O'Keefe [1986b] puts it another way by stating that there is "no explicit next-event set" in the AS CF. Therefore, the time scan determines the time of the next event by scanning t-cells without attempting to identify which activity is due next or which object is causing the next state change [Pidd 1984].

Within the context of the simple single server queueing problem, some confusion exists among Pidd [1984] and Fishman [1973] concerning the description of the time cells and the time scan. Certainly, some means of identifying an arrival as imminent during the current timing cycle is required. Fishman indicates than an object (or entity) clock of the next arriving object needs to be maintained. According to Pidd, a selection of the minimum time cell will determine the time of the next imminent event. Pidd's description is unclear in that time cells are defined to be attributes of permanent objects (entities). The next arriving object is clearly temporary in that it does not exist for the duration of the simulation. Instead, it may arrive and depart the system before the simulation terminates. Pidd, however, later clarifies this by implying that arrivals can be handled by permanent pseudo-objects (as discussed above) which are "arrival machines" for each type of arriving object.

In this case, as explained by Nance [1987], the time scan checks the arrival clock time for each arrival machine. Once system time has overtaken a local arrival clock time, this local clock time for a particular object type may be discarded. Thus, this will enable reassignment of the arrival clock to the next arrival time for a particular arrival machine. By searching the time cells of all permanent (including pseudo) model objects for the next

minimum time value, one can then easily determine the time of the next event and if the next event is an arrival. Such a characterization of the time scan is consistent to that discussed by Tocher [1963] in which he suggests that every time variable be associated with a machine which may be real or imaginary. His example of the single server queueing problem includes an arrival machine (imaginary) and the server (real). The time attributes for these "machines" would correspond to Fishman's object (entity) clocks and to Pidd's time cells.

2.4.2 The Activity Scan

Kiviat [1969] describes the activity scan as a "search" method in which the "scanner examines system state data to determine whether a state change can take place." It is the activity scan that moves the model from event to event because "no event list is maintained" as in the event scheduling world view [Neelamkavil 1987]. It is important to note that the activity scan repeats until no further state change or activity is possible. The condition blocks shown in Figure 2.3 represent the testheads and are checked in order (1 to N). Pidd [1984] emphasizes that the ordering of the activity descriptions (conditions and action blocks, 1 to N) by priorities is very important in accurately representing system behavior by the model. An example is given [Pidd 1984] to illustrate the effect of scan priorities on model results. When the conditions of a testhead are satisfied, the actions associated with it are executed. If a testhead is not satisfied, the next testhead in turn is checked. Activity testheads may be satisfied when a set of boolean conditions becomes true. For an end or finish activity, its testhead is satisfied when a check of t-cell values indicates that the activity is "due". Indeed, Kreutzer [1986] describes the activity scan as a scan of "temporal and other conditions." Thus, activities are associated with a

pure set of conditions that state "when an activity of a given class may start, and [in one form or another, through object structures or some link to the activity] a time cell that specifies when it may finish."

At the completion of a single scan, if one or more actions blocks have been executed, the check of all testheads (beginning at the first) is repeated. The scan continues in this way until no testheads are satisfied. This means that no more state changes are possible for the current timing cycle. If termination conditions hold, the simulation ends. Otherwise, the time and activity scan sequence is repeated.

2.5 The Three-Phase Approach (TPA)

The Three-Phase Approach is a modification of the AS CF which attempts to improve execution efficiency by recognizing that some activities will occur at known and determined times in the future. In the case of such activities, there is no need to test for the satisfaction of certain conditions. The approach is attributed to Tocher [1963] who categorized activities as B-activities or C-activities. A B-activity is one which is bound to a certain object or "machine" and its time of execution is known. For example, in the simple single server queue problem, the next arriving object or the arrival machine is "committed" to an arrival activity which will occur at a determined time in the future. In practice, the binding of the activity to an object may be accomplished by assigning an integer (which identifies the bound activity) to an appropriate object attribute. The object remains committed and the activity is "engaged" or bound by that object as long as the system time is less than that object's local time clock. When system time reaches this local clock time, the object becomes "available." B-activities are due and executed when the object to which they are bound becomes available. The end or finish activity

(discussed in the AS CF) is another example of a B-activity. As alluded to earlier, the traditional repetitive activity scan and testhead checks do not need to be done on the B-activities. In a sense, the B-activities are implicitly scheduled, and the TPA could be said to incorporate a next-event set or the events list approach that characterizes the ES CF [O'Keefe 1986b]. The C-activities can be described as cooperative activities in that dependencies with other activities exist. C-activities with testheads must enter the usual, repetitive activity scan. [Tocher 1963]

In a later paper, Tocher [1965] provides more details on the Three-Phase Approach. Tocher indicates that a convenient way to determine the pending or imminent bound activities is to scan a list of "returning" objects (or entities) which is created at each timing cycle. The bound activities are executed in the order that they are determined from the object list. Thus emphasis is placed on the object and an ordering of objects is implied. Such an ordering could be based on priority (or other technique) and will affect the behavior of the model in the same manner that ordering of the activity scan influences models under a pure AS CF.

ECSL [Clementson 1966,1978] was an early implementation of the TPA. Pidd [1984] provides an excellent overall description of the TPA. Other comprehensive descriptions of the TPA are available in current literature [Crookes 1982; Crookes et al. 1986; O'Keefe 1986a, 1986b; O'Keefe and Davies 1986; Davies and O'Keefe 1987]. Although Tocher [1963] is attributed with the TPA, an earlier description of the approach exists [Tocher and Owen 1961].

With activities now categorized as B or C, the executive can be modified to display the three phases. Figure 2.4 graphically represents the implementation of the three phases:

- Phase A — time scan
- Phase B — execution of B-activities which are now due
- Phase C — activity scan on C-activities

An explanation of Phases A and C is not needed since they follow the time and activity scans of the activity scanning world view. The execution of Phase B is covered above. Therefore, in summary, instead of an activity scan of all activities with their testheads, the B-activities are removed to be executed at their appropriate times. The length of the pure activity scan (which now deals with only the C-activities) is reduced. Execution efficiency is improved.

2.5.1 Variants of AS-based Executives

Interestingly, Tocher [1965] sets forth three possible organizations for activity scan-based executives. First, of course, is the organization which is the pure activity scan and treats all activities as C-activities. This leads to a great deal of "unnecessary testing." Secondly, Tocher suggests an organization which consists of only B-activities. An extreme burden is placed on the modeler using this scheme in that the modeler can no longer present the conditions for an activity in one set of statements. Instead, the modeler must "generate for each entity [object] which has reached the end of a previous activity the appropriate extra conditions, and do this for every participating entity [object]." The third organization, a combination of both B and C activities, represents the Three-Phase Approach discussed above.

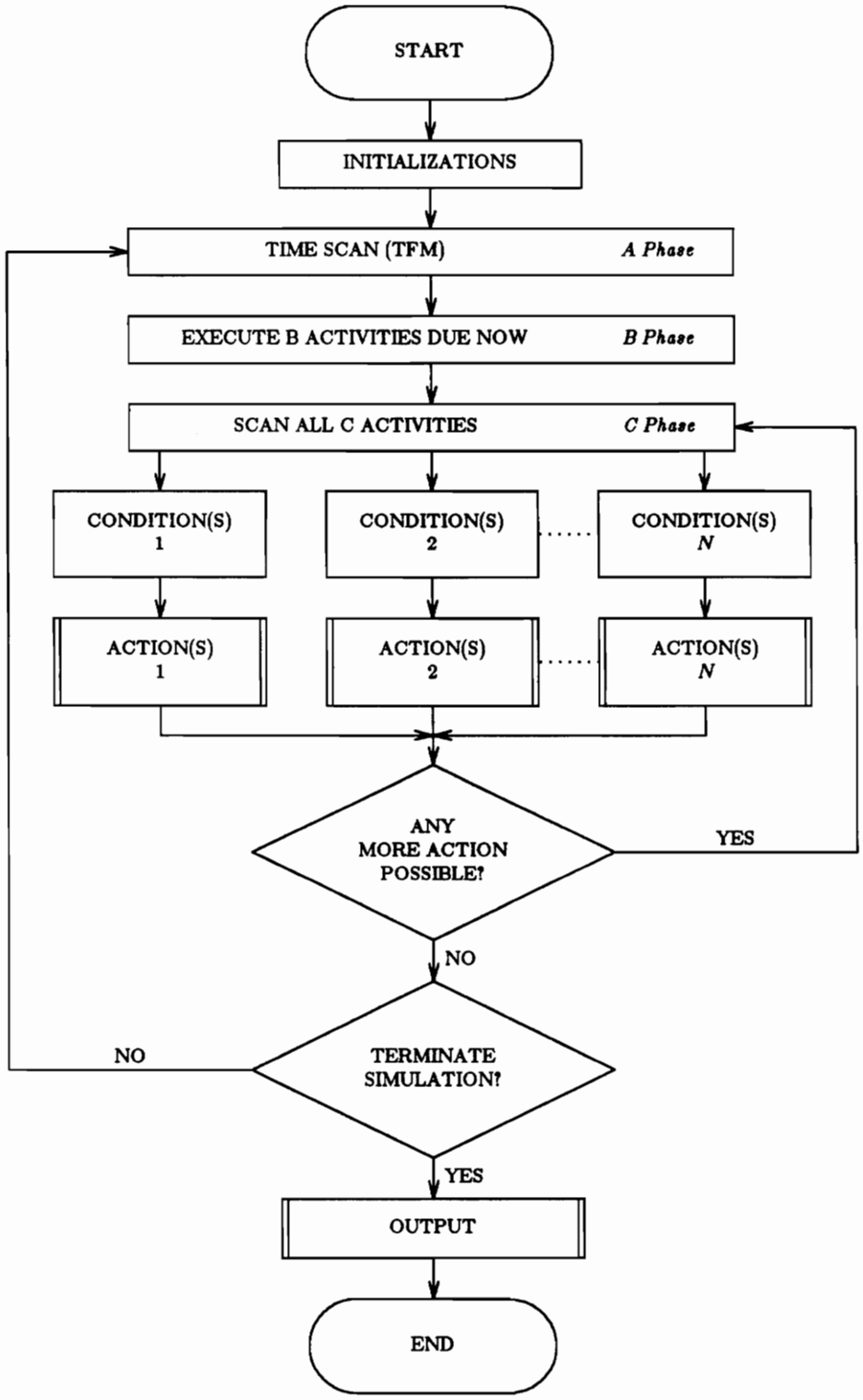


Figure 2.4 The Three-Phase Approach (TPA) Conceptual Framework [Balci 1988]

2.5.2 The Cellular Approach

Closely related to the Three-Phase Approach is the Cellular Approach [DeCarvalho et al. 1976]. At any given time, the model activities can be grouped together into a "cell" with "those entities [objects] which at the current time are already engaged to, or are waiting to be engaged to, one of the activities in the cell." Objects may belong to different cells throughout a simulation as they become engaged in different activities. The grouping of activities in a cell (the basis for the cellular structure) is such that the execution of activities within a particular cell at an instant in time does not depend on state conditions or variables that exist in other cells at that same instant. Therefore, each cell can be considered independent and "non-overlapping" with the other cells of the model. Furthermore, each cell is a grouping of B and C-activities and is essentially a Three-Phase simulation in its own right.

Objects become associated with different cells as time progresses. Based upon this knowledge of an object's process, the model executive is able to discriminately choose which cell or group of B and C-activities should be scanned for execution (the B-activities of interest) and tested (the C-activities of interest) for possible execution. In addition,"there is no point in testing a C-activity within a cell unless that cell has had a B-activity executed" within the last clock cycle. In other words, no state changes have occurred which would alter the testhead conditions of subsequent C-activities within that cell. In this way, the cellular structure provides the ability to eliminate the unnecessary testing of certain C-activities. For large models in which there are many independent events resulting in a large number of cells, this provides further savings in execution time beyond that obtained in the Three-Phase Approach.

2.6 Process Interaction (PI)

Kiviat [1969], Fishman [1973] , and Pidd [1984] provide an excellent overview of the PI CF. Instead of the event or activity, PI uses the process as its basic building block. From the earlier definitions, the process can be considered to be a life-cycle for an object. It represents a sequence of events and interspersed activities through which the object moves. As the object moves through its process, it may experience certain delays and be blocked in its movement. Those delays which are time-based and unconditional (e.g., service times, arrival times) must be handled using future event set algorithms [McCormack and Sargent 1981] and techniques like those used in the ES CF for the determination of the next event time. Those delays which are state-based (e.g., wait-until situations) require a scan of conditions to determine the time(s) at which such delays should be resolved. Therefore, the PI CF combines certain aspects of the AS and the ES CFs while producing an altogether different approach.

Because an object experiences periods of activity (process statement execution) and periods of inactivity (conditional or unconditional delay), one can view an object's process description as being a single set of program statements which act like several different, individual programs. The PI CF enables the modeler to clearly grasp a model's structure now that each object or class of object can be represented by a single, coherent process rather than through multiple event routines. [Kiviat 1969; Fishman 1973]

The PI CF also provides clarity in representing how the various object processes in a model are interacting. When an object experiences a delay in its process and becomes "passive", another model object is allowed to become "active" [Franta 1977] and to start or resume its process. In essence, the object processes within a model behave as coroutines, alternating their executing (or active) status with one another in a controlled

fashion. Kiviat [1969] calls these locations within an object's process (where such delays are incurred and execution is shifted to another object) its "interaction points." In addition, those points at which an object returns to an active state (following such interaction) are named "reactivation points." Another way to view these points are as code locations in the program description of an object's process where it resumes execution following a delay.

Figure 2.5 provides a GPSS-like [Henriksen et al. 1983; Schriber 1974] representation of the PI CF. Each object has associated with it a record of information which includes its reactivation time (if known) and its next reactivation point. Such an implementation, described nicely by Pidd [1984] includes two lists, the current events list and the future events list. We shall call them the current objects list (COL) and the future objects list (FOL) since these lists contain the associated record representations of the objects themselves. In addition, these lists are maintained and used to perform the selection of the next object that will become active. The COL contains objects which are due for activation during the current system time or which are in a wait-until status (waiting for certain conditions to be satisfied). The FOL contains those objects for which a reactivation time is known. Furthermore, the objects in the FOL are most likely ordered by time.

2.6.1 The Clock Update Phase

The ordering of the FOL enables the updating of the system clock in the "clock update phase." The earliest reactivation time from among the objects on the FOL is selected and the system clock is assigned this time value for system time. Then all objects on the FOL which have this time as their designated reactivation time are transferred to the COL, concluding the clock update phase.

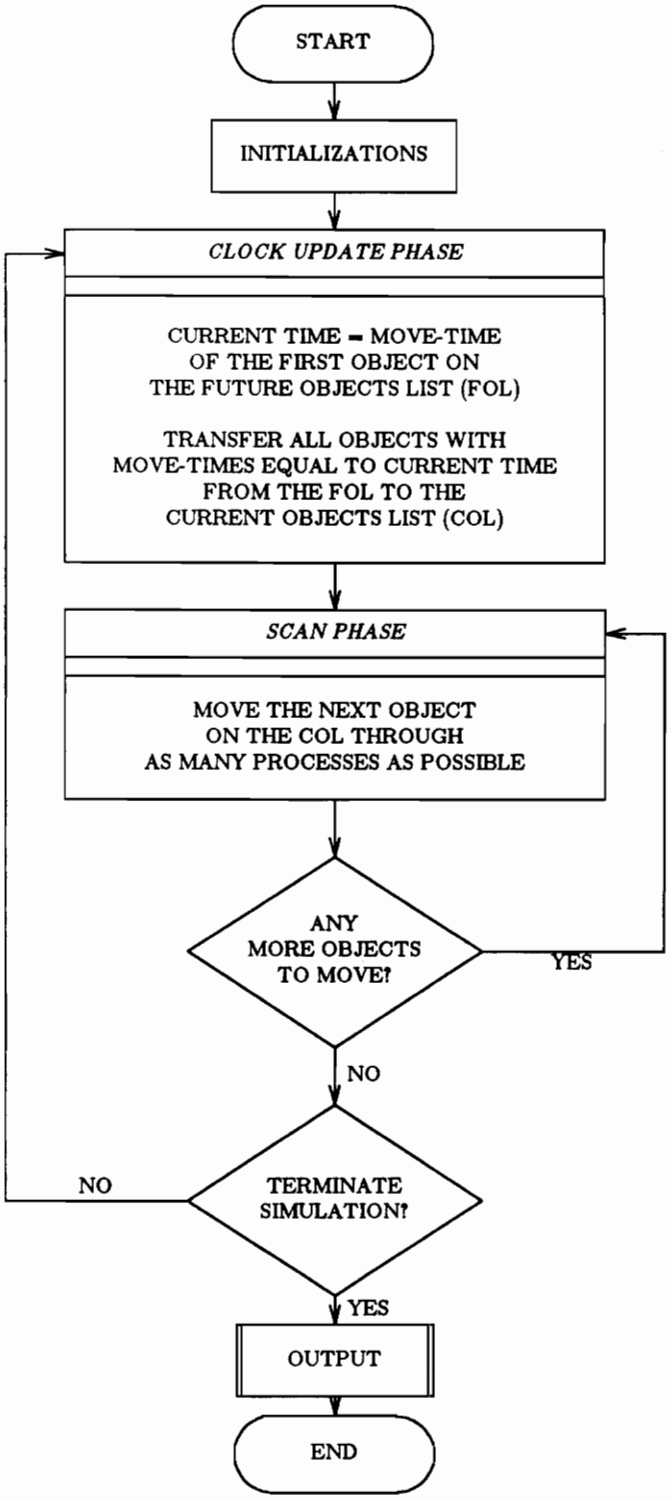


Figure 2.5 The Process Interaction Conceptual Framework [Balci 1988]

2.6.2 The Scan Phase

The "scan phase" next takes each object on the COL, in turn, and tries a restart of its process at its reactivation point. Movement may not be possible for those objects in a wait-until status. If so, such objects remain on the COL. Each such object is moved as far in its process as possible until it is once again blocked or until it completes process execution. If blocked, the object is placed on the COL or FOL as appropriate. The scan phase continues until no further object movement (within its process) is possible. Please note that the clock update phase is restarted following each conclusion of the scan phase (as shown in Figure 2.5) until simulation termination conditions are reached.

SIMULA [Birtwistle et al. 1979; Franta 1977], another process-oriented language, differs greatly from the GPSS implementation approach. Objects are maintained on a single list called the sequencing set, in contrast to the two-list GPSS approach. Unlike the GPSS approach, objects which are in a wait-until status are not directly handled via the single-list implementation. The programmer is responsible to implement the trap conditions which will reactivate an object and place it in the sequencing set.

2.7 Transaction Flow (TF)

TF handles the time and state relationships of the model in exactly the same manner as the Process Interaction CF. However, there are three differences which can be noted.

First, Shannon [1975] used the term Transaction Flow to serve as a categorical designation for simulation languages rather than as descriptive of a simulation strategy or world view. A language of this type uses "specialized blocks" which are assembled into the model's structure. The block diagram which can be easily formed from the language

blocks represents a clear description of the logic and flow of the system being modeled. "Transactions" are created and moved through the blocks, executing specialized actions that are "associated" with each block. The movement of the transactions causes the simulated time to advance.

Shub [1980] makes a further distinction by pinpointing the block structure and its underlying actions that are performed as the key difference. The block structure generates a rigid structure which limits the "examination and communication" among system components. In addition, as entities (transactions) pass through these blocks, "predefined processes" are activated which are hidden to the modeler. Statement languages, on the other hand, like SIMULA, provide generality and flexibility to the modeler. Lower level primitives are available which allow the modeler describe entity communications at any necessary level of detail. The modeler is not constrained by predetermined blocks and a finer level of component interaction can be obtained.

Finally, a third distinction can be inferred from the term "transaction" itself. Tocher [1965], in his review of simulation languages, characterizes the languages as machine or material oriented. He further defines "transactions" to be the material entities. Kreutzer [1986] gives excellent definitions of these two orientations. In a machine oriented view, servers (machines) are the dominating and active influence in the model. They obtain the material entities (transactions), operate on them, and place them in (or remove them from) queues. Conversely, the material oriented models hold the transactions as the dominant entities. Servers, now passive, are "acquired, held, and released again" by the transactions which flow from machine to machine [Kreutzer 1986]. Therefore, in a transaction flow approach, a material oriented view is held. Processes are described in terms of the temporary objects or entities (the transactions) which dominate

and flow through the model. The permanent entities (machines) become passive [Nance 1987].

From the preceding discussion, "transaction flow" is more appropriately used to categorize a simulation language. However, the term can be effectively used to accurately describe a variation of the PI CF.

2.8 The Object-Oriented Paradigm (OOP)

The paradigm has been described as a "programming style" that views "programs as collections of active components (sometimes called classes or actors) and their patterns of interaction" [Kreutzer 1986]. These active components are the basis for the term *object.* Additionally, the paradigm is viewed as a concept for system design. According to Meyer [1987], "object-oriented design may be defined as a technique which, unlike classical (functional) design, bases the modular decomposition of a software system on the classes of objects the system manipulates, not on the functions the system performs." A functional decomposition is less likely to be resistant to change over time than one which is based upon an object-oriented decomposition. Functions tend to change in order to adapt to changing needs whereas objects remain more or less constant [Meyer 1987].

Whether a programming style or a design technique, the paradigm may be viewed ultimately as a software engineering methodology. A distinction must be made between *method* and *methodology.* (One must also be careful not to confuse the term method in this context with the operations (methods) which an object (in the paradigm context) performs.) A *method* is a means of accomplishing a given task and contains the decisions to be made, how these decisions are made, and the order in which they are made. A *methodology,* however, is "a collection of complementary methods, and a set of rules for

applying them" [Arthur et al. 1986]. Although the paradigm's methods and rules are not distinctly available, the material in the following paragraphs indicate that the paradigm does contain component philosophies, approaches, or characteristics. These taken as a whole suggest that regarding the paradigm as a methodology is an accurate assessment.

The paradigm is a powerful methodology which is characterized by the *encapsulation* of data and operations, an *inheritance* mechanism for developing object hierarchies, a *binding* approach which allows the dynamic change of data types, and, in some cases, the ability (*activation, passivation*) to move objects into and out of various states (active, passive, etc.). The following subsections will review these principal features which are embodied in the paradigm.

2.8.1 Encapsulation

The leading experts seem to agree that encapsulation is a primary ingredient to the paradigm. The implementation of an object's actions and "how its internal data is arranged" are contained within the object [Cox 1986]. Encapsulation has been called the "foundation" of the object-oriented approach and provides several important implications [Cox 1986]. First, encapsulation enables a software system to be more "malleable" and resistant to change. An object can be considered to be "encapsulated", an "armor-plated" entity. As "private data and a set of operations that can access that data", an object by nature thus restricts "the effects of change by placing a wall of code around each piece of data" [Cox 1986]. The use of objects therefore improves the reliability and maintainability of system code. Additionally, by inherent abstraction, the object improves the view of the system by introducing a higher level perspective and promotes reusability of code. Meyer [1987] states that "object-oriented design is the construction of software systems as

structured collections of abstract data-type implementations." In conventional programming styles, the "consumer" of a service must specify the details of how each of the desired operations is performed. Using the paradigm, however, one only need to specify what is desired. The "supplier" can then encapsulate how these operations are performed within the object, hiding the details from the consumer [Cox 1986]. This then allows a radical approach which Cox [1986] calls the "software-IC [software-Integrated Circuit]" approach to building systems. The analogy to the use of hardware ICs (which are easily "plugged" into or removed from an electronic system) is intended. Portions of code then become "reusable". For Meyer [1987], the paradigm is "the most promising technique now known for attaining the goals of extendability and reusability." Related terms are *modularity, abstract data typing,* and *information hiding* [Kafura 1987].

2.8.2 Inheritance

A mechanism for inheritance is another distinguishing characteristic of the paradigm. Inheritance is the "ability to define new objects by expanding, contracting, or modifying the functionality of existing objects" [Kafura 1987]. This is considered perhaps the most powerful characteristic of the paradigm. An object *class* can define the generic distinguishing features of a grouping of like objects. Inheritance is a "tool for automatically broadcasting code to classes developed by different members of a team" [Cox 1986]. New instances of an object class can be easily created. These new instances automatically inherit the attributes of that class definition. Inheritance supports hierarchical structures that are commonly found in the real world and provides substantial benefit to the user by improving his understanding and view of the system.

Beyond encapsulation and inheritance, there is less agreement among the authorities

of the field as to the features embodied by the paradigm. Inheritance itself does not seem to have universal approval as a necessary ingredient for the paradigm. (Some languages, like Ada which have been classified as object-oriented, do not support the concept of inheritance.) Yet Meyer [1987] states that "the inheritance concept is essential."

2.8.3 Binding

How binding is handled is very important to the power of the paradigm. Binding relates to the "time at which an object becomes restricted to a fixed collection of types which it can manipulate" [Kafura 1987]. Early binding is common among most conventional languages. Such binding requires that every data type is known before compile time. "Tightly coupled systems" and "static binding" are related terms [Cox 1986]. With delayed binding (also known as late or dynamic binding), the data types can dynamically change during program execution. This kind of binding is "needed in loosely coupled collections where the consumer's code cannot predict the type of data to be operated on until the code is being run" [Cox 1986]. Dynamic binding thus promotes reusability and flexibility. New "data types can be added over time without impacting working code" [Cox 1986]. These concepts of binding enhance the power of encapsulation and add to the measure of its benefit. Objects communicate with each other through the sending of messages. Messaging impacts the concepts of binding. With encapsulation, the object is sent a message which tells it to perform an operation on itself. When reusing object-oriented code and implementing dynamic binding, the binding "occurs only at this point [i.e., when the message is sent]" [Cox 1986].

2.8.4 Activation and Passivation

A final and often common feature of the paradigm is that of activation and passivation, "the ability to save and restore the state of an object independently from the program's existence" [Kafura 1987]. Activation/passivation "provides an automated way to convert arbitrary objects to a symbolic representation that can be stored ..., and an inverse transformation to regenerate objects given the symbolic representation. The symbolic representation can be transmitted between different processes..." [Cox 1986].

2.9 The Process Graph Method (PGM)

The PGM is derived from the parallel computation model which was suggested by Karp and Miller [1966] and later improved upon by the U. S. Navy [Kaplan 1987; Stevens 1987; Hillson 1987]. The Navy's development of the PGM grew out of its work at the Naval Research Laboratory on ECOS/ACOS (EMSP/ASP Common Operational Software Support) Methodology. ASP stands for Advanced Signal Processor, the Navy's current standard signal processor. EMSP, or Enhanced Modular Signal Processor, is the Navy's follow-on to the ASP. Used primarily for the development of signal processing models, the PGM is believed to be applicable to domains other than signal processing and capable of being used as a representation of other model types. The basis for the PGM is the process graph, a directed graph of *nodes* and *arcs* which is classified as a *data flow* model. Three of the primary benefits of the process graph are its parallel computation capabilities for greater throughput, the ease at which modelers can perform top-down design, and portability of applications.

Each node in the process graph represents a *primitive function* (some type of computation or process) or may alternatively represent a *subgraph* which is itself a process

graph. Such a convention allows the modeler to use abstraction and modularity to represent complex models in a fashion that is more easily understood. A node, when implemented, contains the following:

- logical ports — to handle data flow,
- resident primitive function procedures, and
- a primitive interface procedure (PIP) — to handle data transfer input and output between the primitive functions, queues, and auxiliary data storage entities.

The arcs represent queues which contain the input and output data needed by the nodal primitive functions for execution. The data is ordered on a first-in, first-out basis. Each queue has a *mode* type which characterizes the data type that it carries in residence (integer, real, etc.). A queue may carry trigger pulses for synchronization instead of data for computational purposes. Queues also provide the necessary connectivity for the graph.

Nodes execute only when the data necessary for execution are available at the input queues. Such a node execution scheme allows multiple computations to be performed in parallel thus generating a greater throughput. Not all graph data is transmitted along a queue. *Node execution parameters* (NEPs) and *auxiliary data storage entities* such as graph variables, graph controls, and graph instantiation parameters provide additional graph control information. Process graph execution is guided by a command program which handles input/output to/from the graph, manipulates the control attributes, and communicates with the rest of the world, external to the process graph itself.

The following sections further elaborate on the key elements of process graph representation (primitive functions, NEPs, and auxiliary data storage entities) and

summarize the current literature descriptions of Weitzman [1986] and Stevens [1987].

2.9.1 Primitive Functions

The primitive functions that are represented by the nodes of the process graph perform a computation on the input data elements and distribute the output along the output queues. The primitive functions may be simple or complex in nature. Fast fourier transforms, sort routines, and matrix multiplication routines are examples of typical primitive functions [Kaplan 1987].

2.9.2 Node Execution Parameters

Threshold, offset, read, consume, and *produce* are the NEPs that specify how the data elements are manipulated on the input and output queues. NEPs may be constant ("fixed") during graph life or they may be recomputed ("variable") for a particular graph node. The *threshold* is the parameter which dictates the number of data elements that must be on each input queue before the node can be executed. Data elements can be skipped and bypassed when reading in data from an input queue. A number quantifying this is referred to as the *offset.* The *read* and *consume* NEPs are related. *Read* refers to the number of data elements which are to be delivered to a node. *Consume* indicates the number of elements which are to be removed from an input queue. Finally, *produce* specifies the number of data elements that is output or written to the output queue.

2.9.3 Auxiliary Data Storage Entities

A graph variable (GV) may be *internal* or *dynamic.* An internal GV is local to the graph in which it is declared. All nodes within a specific graph have read/write privilege

to local GVs. Internal GVs may be passed to subgraphs as control. As such, nodes within these subgraphs have only read privilege. A dynamic GV is defined in the command program and is passed to a graph as a control. In this case, the graph has only read privilege.

Graph controls (GCs) are GVs which are passed from the command program or from a graph to subordinate graphs. The originator of the GC may change its value for subsequent execution sequences. Graphs in receipt of the GC have only read privilege and read the latest value of the control at each invocation.

Graph instantiation parameters (GIPs) are constants which are passed by value to the graph at its instantiation or definition. The GIPs are primarily used for establishing the start time constraints on the graph.

2.10 The Entity-Relationship Model (ER) and ER Approach (ERA)

The ER model [Chen 1976] is a data model which is based upon set and relation theory. Of the other three major data models listed by Chen [1976] (network, relational, entity set), he claims that one "may view the entity-relationship model as a generalization or extension of existing models" and that the ER model "has most of the advantages of the above three models." Since the ER model represents an encompassing data model, it is worthy of review as a conceptual framework with potential application to model representation in discrete event simulation. Chen [1976] represents the definitive work which introduces the ER model. This brief description attempts to summarize the principal components and distinguishing features of the ER approach (ERA) to modeling through the use of the ER model. Claimed advantages [Chen 1976] include:

- "adopts the more natural view that the real world consists of entities and relationships"
- achieves "a high degree of data independence"
- "incorporates some of the important semantic information about the real world"

Chen [1976] introduces the ER model through the context of levels of logical views of data, and develops the ER model for two of these levels which are defined as follows:

- Level One (a *conceptual* level) — "information concerning entities and relationships which exist in our minds" and
- Level Two (a *representational* level) — an information structure or organization of information in which data represents entities and the relationships which exist among them.

The next two sections cover Chen's development of the ER model at these two levels.

2.10.1 ER Model Development at Level One

Here we are concerned with the conceptual view (of entities, their relationships, and their values) as it exists in our minds. To fully develop our understanding of the model representation at this level and to avoid confusion, the following relevant terms from Chen [1976] must be defined for the ERA:

- *entity* — a "thing" having discriminating features which allow it to be distinguished among others.
- *relationship* — an interdependence, bond, or "association" among entities.

An entity may be a "specific" student, professor, or university, for example, whereas

student-professor represents a relationship between student and professor entities. The term *role* refers to the function that an entity takes within a defined relationship [Chen 1976].

Entities may be grouped into *entity sets* upon which predicates can act to test for set membership of an entity. Entity sets such as STUDENT and PROFESSOR have members which are student and professor entities. The ER model suggests a correlation to the OOP, especially the inheritance mechanism. If a specific entity is a member of an entity set, we automatically know that it possesses the traits which are "common" to the other set members.

Relationships may also specify similar groupings called *relationship sets*. A relationship set is a "mathematical relation among n entities each taken from an entity set..." In other words, the members of the relationship set are relationships that are formed by tuples of n entities, where each entity is a member of some entity set. The relationship set, DEPT-PROFESSOR consists of 2-tuples derived from entities which are members of the DEPT and PROFESSOR entity-sets such as [CS, Miller], [ECON, Kenyon], etc.

Entity and relationship information is maintained within sets which contain values (called *value sets*). Value sets such as CREDIT-HOURS, GRADE, and COURSE-ID might contain the values 3, B+, and 4150. Entities and values are linked to one another by *attributes*. Attributes are functions which map from "an entity set or a relationship set into a value set or a Cartesian product of value sets..." For example, the entity set COURSE might have an attribute COURSE-DESCRIP that maps into the value sets of DEPT-ABBREV, COURSE-ID, and CREDIT-HOURS. In addition to entities, relationships can also have attributes.

To organize this conceptual information, Chen [1976] proposes that the entity and relationship information be separated in order to enable the identification of the "functional dependencies among the data." In addition, a tabular representation or structure is helpful to relate the ER model to the relational model. In the tabular representation for entities, all row information relates to a single entity. A particular column holds values from a particular value set. Attributes are represented as column headings, and one or more columns may apply to a single attribute. In other words, in keeping with the definitions above, the attributes map a row identifier (particular entity from an entity set) into one or more value sets (grouped columnar information). Relationships can also be organized in a similar fashion [Chen 1976].

2.10.2 ER Model Development at Level Two

At this level, Chen [1976] uses the ***primary key*** which is a unique identifier of specific entities (or relationships). The use of primary keys moves the model representation from one which is purely conceptual to one where the conceptual objects exist with a "direct representation of values." The primary key serves as this "direct representation" of an entity or relationship. Attribute values are used as primary keys if their mapping function is one-to-one. In many cases, more than one attribute or a group of attributes (an ***entity key***) is needed to find such a one-to-one mapping for the members of entity sets. In some cases, it is not possible for an entity to be uniquely represented solely by its own attribute values. Instead, a usable primary key may be contrived or the primary key of another entity which participates in a relationship with the entity of concern is used. The primary key of a relationship is composed from the primary keys of the entities that make up the relationship.

The information at this level can now be organized in a tabular representation, as before, which identifies the specific entities by their primary keys. Such a table (an *entity relation*) is composed of rows of values called *entity tuples* [Chen 1976]. When a relationship is used to identify an entity, this is called a *weak entity relation.* Otherwise, it is a *regular entity relation.* In a similar manner, tabular representations of relationships (*relationship relations*) which are composed of rows of *relationship tuples* [Chen 1976].

2.10.3 Using the ER Model

Four steps [Chen 1976] to using the ER model (particularly for database design) and applying the ERA follow:

- "identify the entity sets and the relationship sets of interest"
- "identify semantic information in the relationship sets such as whether a certain relationship set is a 1:n mapping"
- "define the value sets and attributes"
- "organize data into entity/relationship relations and decide primary keys"

Therefore, successful application of the ERA relies on the proper **identification** of set membership and semantics, **definition** of value sets and attributes, and **organization** of data into relations.

2.10.4 ER Model Classifications

Chen [1983] proposed a framework for classifying ER models which is based on the "capabilities and limitations " of each model's *relationships* and *attributes.* The two broadest classifications of ER models concern their treatment of relationships. When relationships can be "defined on more than two entities" the model is classified as

Generalized (N-ary) Entity-Relationship Models (GERM). If relationships are only allowed on two entities, the model is a Binary Entity-Relationship Model (BERM). These categories can also be further classified by their treatment of attributes. There are three possibilities:

- attributes allowed for both entities and relationships
- attributes allowed for entities only or
- attributes not allowed.

Chen [1983] believes that ER models under the different classification categories may be converted from one form into another. This implies:

- ability to model a system by "favorite" model form and then translate it to other forms for the purpose of "presentation to others."
- ability to implement a system which supports "several types of entity-relationship models."
- ability to demonstrate equivalence between different ER models.

2.11 The Entity-Attribute-Set (EAS) Approach

Markowitz et al. [1983] claim that the EAS approach is derived from CSL, GASP and SIMSCRIPT simulation languages which were introduced in the early 1960's. The terms ***entity, attribute***, and *set* were central concepts that were consistently defined for each of these languages as follows [Markowitz et al. 1983]:

- entity — "some concrete or abstract 'thing' represented by the simulation."
- attribute — "some property or characteristic of the entity"

- set — "an ordered collection of entities."

Markowitz et al. [1984] provide an excellent description of the EAS approach to system modeling. In the EAS approach, entities of interest in the system are kept track of by a database. Within a system such as a naval task force at sea, these entities might include surface ships, submarines, planes, missiles, and torpedoes. The database also maintains complete information on a particular entity to include attribute and value data, the identification of the sets to which the entity belongs or which it owns, and the membership of the sets which it owns. Attributes of a submarine could include course, depth, and speed with respective values of 090 (degrees), 150 (feet), and 25 (knots).

A key characteristic of the EAS approach is that sets are *ordered* as noted above. This ordering may be strictly on a first-in, first-out (FIFO) basis or in some other determined order. Considering the task force example, this system may contain the set of all combatants ordered by hull designator and number. The task force model itself would likely be the owner of such a set. Each combatant may then also own a set of weapons perhaps ordered by lethality or protection capability to the owning platform. Hierarchical decompositions and tree-like structures of the system are thus easily defined.

The EAS approach to system description "combines an 'object-influenced' static view with an 'event causality' dynamic view [Nance and Overstreet 1986]." Set ordering provides the ability to represent timed events and a system state can be represented in a database. A function can be determined which transforms the database from state to state. Yet, such a transformation cannot be clearly depicted apart from the model implementation. Thus, model dynamics are difficult to represent.

Markowitz et al. [1983,1984] raise interesting comparisons between the EAS and ER approaches. The EAS approach is very similar to the ERA in that it allows the modeler

to more naturally represent the system of concern. Entities, attributes, and relationships are the cornerstones of the approaches. The EAS approach is inherently able to contain more information than the ERA due to the set ordering. Yet it is difficult to describe many-to-many relationships in the EAS approach.

2.12 The Conical Methodology (CM)

Aimed at assisting the modeler during the model development phases of the model life cycle [Nance et al. 1984; Balci 1986], the Conical Methodology (CM) is *a practical guide* for accomplishing the model development tasks, particularly model definition and specification. Furthermore, the CM is based on the object-oriented paradigm in that the world is viewed as being composed of objects (model/submodel components) which are distinguished by unique features or *attributes.*

According to Nance [1986], the CM seeks to achieve five primary objectives (model correctness, testability, adaptability, reusability, and maintainability) through the effective use of several key principles. Citing from his work, these principles generally "state *how* the objectives of the CM are to be achieved and at the same time *what* is needed so that the development process can realize those objectives." Top-down definition and bottom-up specification techniques are at the core of the procedural guidance that is derived from and prescribed by the CM principles. This description emphasizes these techniques and discusses them in further detail since are they central to the CM and the guidance which it provides as a conceptual framework to the modeler. The CM principles which underlie these techniques and which are not covered here are fully discussed by Nance [1986]. Of interest, the CM has provided the fundamental guidance for the development of three prototypes of the Model Generator tool of the SMDE

[Balci and Nance 1987a, 1987b]. Bottom-up specification is, however, only supported by the third prototype which is described by Barger [1986].

2.12.1 Top-down Model Definition

Top-down model definition under the CM is accomplished through a hierarchical decomposition of the model into successive submodels. At each level of decomposition, *attributes*, including attribute dimensionality and range of values, are assigned (to the particular submodel associated with that level) and are classified by type in accordance with Nance's [1986] taxonomy tree. Nance [1981a] exhaustively defines all elements of this tree which is a hierarchical decomposition of the CM attribute types. Summarizing briefly, attributes may provide direct knowledge or information about a submodel (*indicative*) or they may relate a submodel to other submodels (*relational*). Relational attributes may be further classified as *hierarchical* (establishing a "subordination" of one submodel to another) or as *coordinate* (establishing a "bond or commonality" between two submodels). Indicative attributes of a submodel have values assigned once (*permanent*) or more than once (*transitional*). Transitional attributes are classified as *status* (value assignments occur from a limited set of value) or as *temporal* (values are a function of time). The product of the top-down definition stage is a static model representation.

2.12.2 Bottom-up Model Specification

In general, "specification is the process of describing system behavior so as to assist the system designer in clarifying his conceptual view of the system" [Barger 1986]. Model specification requires an "indepth recognition of the interactions among attributes, particularly as these interactions vary with time" [Nance 1981a]. The modeler must "specify"

these interactions with expressions which determine the value assignments to the attributes. The specification task, once completed at some level in the decomposition hierarchy, is performed at successively higher levels until there are no further levels to be specified (ie.,the top model level has been reached). In contrast to model definition, the specification produces a model representation which contains the necessary information for model dynamics. In addition, the "typing, dimensionality, and value information supplied by the modeler enable subsequent diagnosis for consistency (type) and correspondence (dimensions and value range)" [Nance 1986].

Bottom-up model specification is accomplished by beginning at some base-level submodel. The determination of the beginning point for the specification task is an area of current research. Nance [1981a] suggests that one might select the submodel which has the most assigned attributes. Research by Barger [1986] promotes two possible approaches, the "basic method" and the "status attribute approach.". (Barger's work, while based upon the application of the CM, was focused upon creating a specification which would result in a condition specification. The conclusions derived from this research can be applied generally to the application of the CM to derive a useful specification.) The basic method prescribes the selection of some submodel attribute, the specification of the condtions and actions that result in changed values for that attribute, and the repetition of this process for each submodel attribute in the model. The status attribute approach is an "extension of the basic method..." [Barger 1986]. It begins with the selection of some status attribute for specification and leads to the full coverage of all status attributes. Such a method could also be applied to both temporal and permanent attributes. Barger [1986] suggests that the status attribute approach "...shows potential for providing more structure and guidance to model specification..." than any of the other

approaches.

2.13 Structured Modeling (SM)

The Structured Modeling approach [Geoffrion 1987a,1987b,1987c] is a bold attempt to provide not only a generic *framework* for model representation but also an *environment* to meet total model developmental needs throughout the model's life-cycle. Many similarities exist between the representation characteristics of this approach and the Conical Methodology. Therefore, SM can be used in a top-down model design strategy which embodies a similar stepwise refinement approach and which results in a well documented, easily communicated design. Life-cycle objectives via an interactive environment are also strongly akin to the primary goals of the SMDE (Simulation Model Development Environment) research project [Balci and Nance 1987a]. The SMDE is currently being directed toward the discrete event simulation domain. However, SM aims to cover the same plus other major modeling areas (mathematical programming, database theory for data models, conceptual graphs and knowledge representation, graph grammar-based systems, etc.) [Geoffrion 1987a]. The issues of SM which pertain to model representation will be reviewed and described due to their applicability to discrete event simulation and to our notion of conceptual frameworks. Modeling environment considerations will not be covered.

The SM framework for model representation uses "a hierarchically organized, partitioned, and attributed acyclic graph" for model semantic and mathematical structure. The framework can be decomposed into **elemental**, **generic**, and **modular** structures. Each structure serves a determined aim and can be represented in graphical or textual form. Most of the existing research conducted by Geoffrion has been concerned with tex-

tual representations only. The discussion in the following sections highlights the key elements of each structure and is a summary and paraphrase of Geoffrion's description of the basics of SM [Geoffrion 1987a]. Quotations, unless otherwise specified, are from his work. The reader is encouraged to scan the SM application in chapter 3 in parallel with this section. The SM application provides examples of the following concepts.

2.13.1 Elemental Structure

The construction of an elemental structure [Geoffrion 1987a] is intended to completely capture "the definitional detail of a specific model instance." Within Structured Modeling, five types of model elements are defined. ***Primitive entity*** elements are the model primitives which "represent things or concepts." ***Compound entities*** are similar to the primitive entity elements but are "defined in terms of other things or concepts." Primitive entity elements and compound entity elements have no associated value, and thus differ from the ***attribute***, ***function***, and ***test*** elements which have value. The attribute elements represent the properties of model components and have constant value. When an attribute element is subjected to the control of a "solver" or executive, the attribute element may have variable values. The function elements represent the calculable properties of model components and have a value which is variable and dependent upon the values of other model elements in accordance with some known rule. Test elements are similar to function elements but can have only true or false values.

Model elemental structure is generated through the formation of model elements of these types into a directed graph in which the nodes represent the elements. The graph's arcs depict a reference "call" for element definition requirements from the calling element (in the head node of an arc) to the called element (the tail node). Such graphs are acyclic

in that circularity of definition is not desired. Finally, the graphs are attributed (node and arc attributes) to represent element values, calculation rules for test and function elements, and the ordering of nodal arcs.

Most often, the elemental graph is much too detailed and is therefore difficult to represent in a manner which is easy for the modeler to understand. Therefore, it is common to use a table representation of the elemental structure. Such tables are called *elemental detail tables*. Elemental detail tables contain instance data and low-level model information which is necessary for a complete model specification.

2.13.2 Generic Structure

The generic structure accomplishes [Geoffrion 1987a] the grouping of elements according to "natural familial" boundaries. In effect, such a grouping is a partition in the mathematical sense where each partition is a "cell" or "genus" of elements which have "generic similarity." Generic similarity among elements means that "every element in a genus calls elements in the same foreign genera." The generic structure thus provides the modeler with a natural view of the system under study. By identifying and naming the element groupings or genera, the elemental structure graph (elemental graph) can be converted to the generic structure graph (genus graph).

2.13.3 Modular Structure

The modular structure [Geoffrion 1987a] is a further refinement on the generic structure. The modular structure is created in order to bring into play the concepts of data abstraction and information hiding. "Modules" are formed by grouping the genera "into conceptual units ... according to commonality or semantic relatedness." Modules, them-

selves, can then be grouped into higher order modules. In this way, complex models can be simplified into a representation which will be better understood. The modular structure is essentially a rooted tree. The leaves of the tree are the genera and the interior nodes (the modules) represent the "conceptual units comprising their descendent genera." The entire model is represented by the root node within the modular structure.

Some final requirements must be maintained by the modular structure. It must be capable of being placed into an indented list, "textual" representation. When such a representation corresponds to the preorder traversal of the modular structure, it is called the *modular outline*. Furthermore, this type of modular structure is called a *monotone* if it does not include forward references. This monotone requirement is critical so that a solver can progressively and effectively update model information on a single pass through the modular outline. The acyclicity which is maintained via the elemental and generic structure helps to determine this monotone ordering [Geoffrion 1987c].

In summary, the ***structured model*** is then an "elemental structure together with a generic structure satisfying similarity and a monotone-ordered modular structure [Geoffrion 1987c].

2.14 Condition Specification (CS)

There has been a recognized need for effective tools which will support model specification and documentation. Nance [1977] suggests a Simulation Model Specification and Documentation Language (SMSDL). An SMSDL would facilitate the construction of the model specification, encourage model documentation, assist in bridging the communication gap between modeler and customer, and produce a precise, yet sufficiently general, model description which is independent of existing simulation programming languages

[Barger 1986]. The CS is an SMSDL, suitable for use in a Model Management System (MMS) [Nance et al. 1981], which produces a model specification that can be analyzed to:

- "detect potential problems with the specification"
- "assist in the construction of an executable representation of the model" and
- "construct useful model documentation." [Overstreet and Nance 1985]

The CS, attributed to Overstreet [1982], formalizes the time and state relationships of the model by the use of a set of language primitives. The resulting formalism enables the precise expression of the model's static and dynamic character and the separation of the specification of the dynamics from that of the data. Furthermore, the CS provides a representational foundation upon which additional analysis can be conducted for efficient model implementation [Overstreet and Nance 1985]. The CS does not dictate the time flow mechanism to be used in building the model. The following sections discuss the principal components of the CS: the *interface specification*, the *specification of model dynamics*, and the *report specification*. See [Overstreet and Nance 1985] for a complete description of the primitives which are used to describe and implement the above components.

Overstreet and Nance [1985] describe a Pascal-like form of the CS which does not specify a complete syntax. This is not critical in that the CS may take other forms, as long as these other forms define model behavior without ambiguity.

2.14.1 The Interface Specification

The input and output attributes of the model are described within the interface specification [Overstreet and Nance 1985]. These attributes completely specify the communication or transmission links between the model and its surrounding environment.

The input and output attributes are identified in the specification by name and are typed (input or output, and data type).

2.14.2 The Specification of Model Dynamics

The specification of model dynamics [Overstreet and Nance 1985] consists of a set of *object specifications* and a *transition specification.*

The object specification represents a complete listing of all objects (by variable name) and the identification of all attributes for each object. A value range is given for each attribute. The object specification must contain a "special" object, *environment.* The environment object's attribute listing includes *system time* and any existing model inputs.

The transition specification contains the description of model dynamics in the form of condition and action pairs (CAPs). The condition portion of each CAP is simply a Boolean expression "composed of standard operators, model attributes, and the special sequencing primitives WHEN ALARM and AFTER ALARM" [Overstreet and Nance 1985]. The WHEN ALARM primitive allows the description of a *determined* condition (see Section 2.3). The AFTER ALARM primitive enables the specification of actions that depend on a combination of time and other attribute conditions, in a compound condition expression. Overstreet and Nance [1985] call these *mixed* conditions. *Contingent* conditions are represented by Boolean expressions which do not utilize the WHEN ALARM and AFTER ALARM primitives. The action in each CAP can be any one or a combination of the following five action types: changing attribute values, sequencing time, creating and destroying objects, producing output, or terminating an instantiation of a specification. Initialization and termination pairs must be included in the set of CAPs in the transition

specification.

2.14.3 The Report Specification

The report specification is defined for the data output or results of model execution. Overstreet and Nance [1985] do not prescribe a form or syntax for the report specification but suggest that one could use CAPs as in the transition specification.

2.15 System Theoretic Approach (STA)

The STA [Zeigler 1976, 1984a] is an approach to model definition and specification which contains the ability to identify the static and dynamic structure of the model, uniformity and strict hierarchy in the definition of the static structure, and the ability to define a range set for variables of the model. The STA is based on set theory and the systems modeling formalism and provides a comprehensive, yet general, model representation.

The *system* is the basis for model description and is "a collection of interacting component systems" [Zeigler 1984a]. Since this is a recursive definition, hierarchical decomposition of the model is possible. The incorporation of the ideas and principles of set theory also allows abstraction in this approach.

A system model can be informally represented by describing its *components, descriptive variables,* and *component interactions.* Zeigler [1976] defines these as:

- components — "the parts from which the model is constructed"
- descriptive variables — "tools to describe the conditions of the components at points in time" and

- component interactions — "the rules by which components exert influence on each other, altering their conditions and so determining the evolution of the model's behavior over time."

The following paragraphs summarize the STA and introduce its key concepts and terminology [Zeigler 1976, 1984a].

2.15.1 Preliminary Concepts for Formal Model Specification

The definitions above can now be used to introduce a formal model representation. Initially, we consider only models which are ***autonomous***, having no input variables. The descriptive variables make it possible to fully describe the condition or state of the model at any given instant in time. A ***well-described*** model is one in which the descriptive variables at time t can determine those at time t' in the future. The ***state variables***, a subset of the descriptive variables which do not include the input variables, can be used to accomplish this mapping without having to know the values of all descriptive variables. Only values of the state variables need to be available at time t for all values of descriptive variables to be computed (using the rules of component interaction) at future time t'.

The rules of component interaction are represented by the ***state transition function***, δ, and the ***output function***, λ. The arguments to δ are the values of the state variables at some time t which δ maps to a corresponding list of values of the state variables at time t'. The output function, λ, then takes the results of δ as its arguments and produces a list of values for the ***output variables*** which may be the complete set of descriptive variables or a subset of the same. The values of the output variables which are produced by λ represent the status of these variables at time t'. The output variables are those model variables (from among the descriptive variables) which the modeler is interested in

tracking.

The *state* of the model at any time instant t can then be defined as a single list of values, each value being determined (in accordance with the transition function) from the range set of its corresponding state variable. In set notation, the cross product of these range sets represents all possible model states. All possible model outputs can be represented in a similar fashion. A single model *output* would therefore be a single list or element of the cross product of the range sets of the output variables.

An ordered sequence of states (generated by model execution) can be mapped to a corresponding sequence of associated times. This mapping is called the *state trajectory.* An *output trajectory* maps a sequence of outputs to their associated times. The set of the state trajectories and the set of output trajectories specify the state and output *behavior* of the model.

With the consideration of *input* variables, "variables whose values are determined externally to the model" [Zeigler 1976], the concepts and notation change slightly to accomodate the case of *nonautonomous* models. However, the underlying principles remain the same. Descriptive variables are now composed of the input and *non-input* variables. When the values of the non-input variables at time t and the trajectory of values of the input variables over some time interval t to t' can be used to derive the values of the non-input variables at time t', the model is *well described.* As before, if this mapping can be performed with only a subset of the non-input variables and their values, then this subset is the set of *state variables.* The state transition and output functions are no longer functions of a single variable. The state transition function, δ_h, now maps state **and** input values at time t to state values at time t'. The output function, λ, likewise maps the resulting state values and the input values to output values at time t'. Zeigler

[1976], in describing these nonautonomous functions, considers the time invariant discrete event model, where the time interval $t-t'$ is equal to h, some time constant. In other words, time is stepped into the future at constant intervals rather than random intervals.

2.15.2 The Discrete Event System Specification (DEVS)

The formal specification derived thus far must now be changed to allow time to increment according to the next-event method which is prevalent in discrete event models. Zeigler [1976] refers to the time of the next event as its "hatching" time. DEVS incorporates these needed changes.

The Discrete Event System Specification (DEVS) under the STA is a six-tuple made up of INPUTS, STATES, OUTPUTS, δ, λ, and *ta* when the system is made up of

input variables $\alpha_1,\alpha_2, \ldots, \alpha_n$

state variables $\beta_1,\beta_2, \ldots, \beta_n$

output variables $\gamma_1,\gamma_2, \ldots, \gamma_n$

countdown clock variables $\sigma_1,\sigma_2, \ldots, \sigma_n$.

Each element of the six-tuple is defined as follows:

- INPUTS — a set ,the cross product of the range sets of the input variables
- STATES — a set , the cross product of the range sets of the state variables
- OUTPUTS — a set, the cross product of the range sets of the output variables
- δ — a two-part state transition function

  δ_ϕ — function which maps model state at time t to model state at time t' where t' is the model's next hatching time and where no external events occur.

δ_{ex} — function which maps model state at time t and model input at time $t+e$ to the model state at time $t+e$ where e time units from time t occurs before the next hatching time t'.

- λ — output function to derive outputs
- ta — time advance function which maps a model state (call it s) at time t_i to the model's next hatching time.

  The countdown clock variables are a subset of the state variables and can be represented at model time t_i as a list of values, $\sigma_1, \sigma_2, \ldots, \sigma_n$. The time advance function derives the time interval to the next hatching time which is simply the minimum value of the set of σ_i where σ_i is greater than or equal to zero. Thus, the next hatching time is $t_i + ta(s)$.

The DEVS sets provide the static structure of the model. Model dynamic structure is obtained via its functions. With DEVS, discrete event models can be formally specified. DEVS is developed from a more general formalism, the Systems Modeling Formalism [Zeigler 1984a], which encompasses not only the discrete time base, but also a continuous time base. Only DEVS is presented here since only the discrete event domain is considered.

CHAPTER 3

APPLYING THE CONCEPTUAL FRAMEWORKS FOR MODELING A TRAFFIC INTERSECTION

In this chapter, we apply each CF for modeling the Traffic Intersection (TI), shown in Figure 3.1, located at the intersection of Price's Fork and Tom's Creek Roads in Blacksburg, Virginia. A single traffic light with north, south, east, and west directions controls vehicular traffic in each of the intersection's eleven lanes. The central intersection space is conceptually divided into thirty-five blocks through which the vehicles travel. The blocks in a vehicle's path are used as locators for that vehicle as it moves through the intersection and enable the representation of a smoothly flowing traffic pattern.

Each CF under review has been illustrated in its corresponding section by constructing (to varying levels of detail) a model of the traffic intersection (based on the system of Figure 3.1) in accordance with the CF's concepts and principles. The CM CF application is covered first since it provides a clear description and definition of the TI. This coverage serves as a valuable reference to understanding the TI which is the basis for the subsequent CF representations of the model.

The TI was selected because it offers complexity of model component interaction unlike that found in the usual textbook examples. Reviewing each CF in the context of the TI exposes these CFs to a real world problem. Care is taken to be circumspect in drawing conclusions from these CF applications under such a restricted domain. The

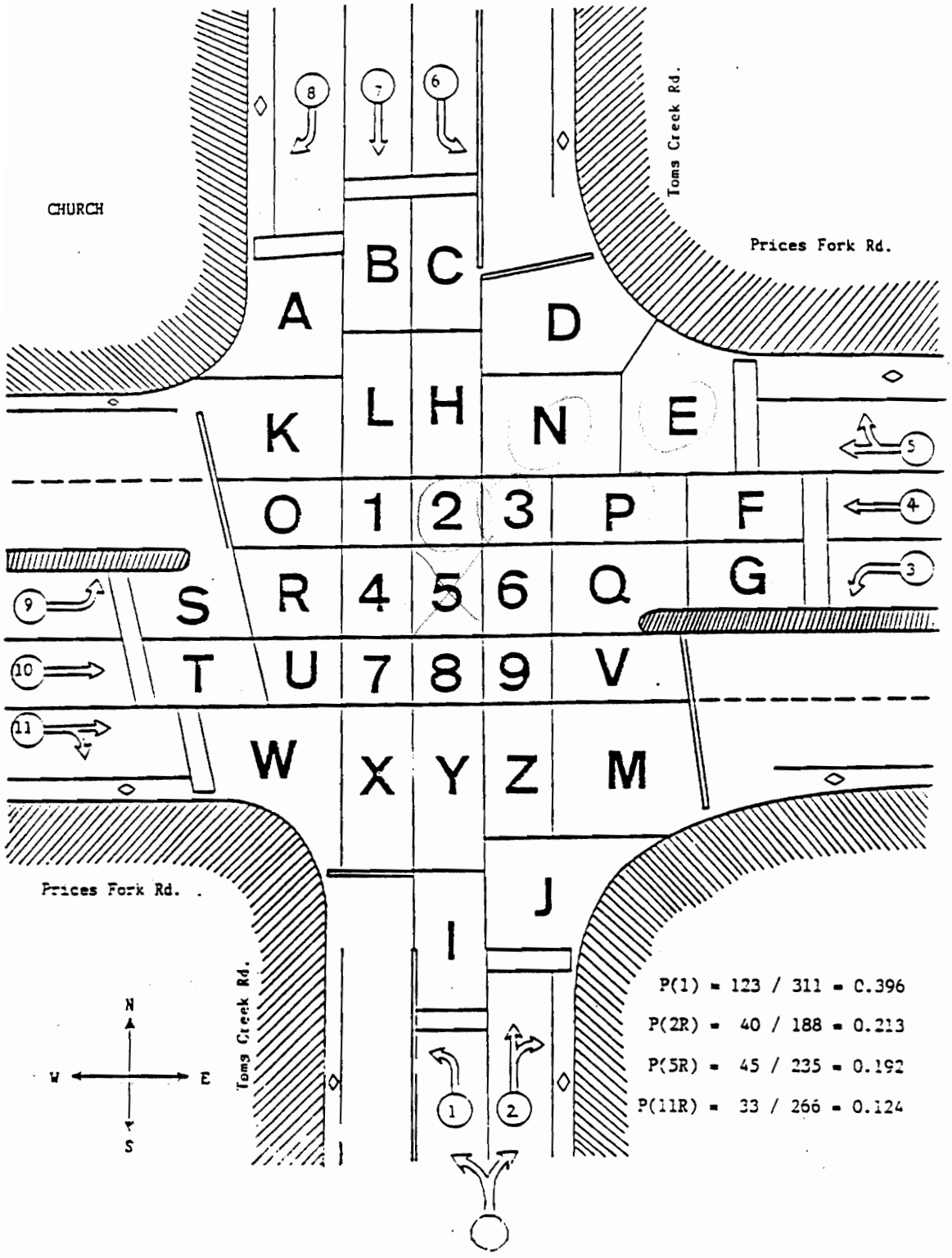


Figure 3.1 The Traffic Intersection (TI) System

intent, however, is to develop a starting point for discussion by accomplishing a thorough application under a single application domain. Future work under other domains is needed.

3.1 Modeling the TI by using the CM

The following text is a definition of a Traffic Intersection Model using the CM outline.

Traffic Intersection Model (CM Representation)

I. Statement of Study Objectives

To compare current light timings with two other alternative ones to determine if average vehicle waiting times can be reduced during "rush hour" traffic conditions between 4:45 p.m. and 5:15 p.m.

A. Definitions

1. N = number of lanes
2. m = number of vehicles departing from lane j (j=1,2,...,N)
3. *Arrival time* = The time at which a vehicle joins the end of the waiting line or the time at which the vehicle's front end moves across the first white line in the lane.
4. *Departure time* = The time at which the vehicle's rear end clears the last white line in the path of travel.
5. IAT_{ij} = Interarrival time of the ith vehicle in lane j (i=1,2,...,m; j=1,2,...,N).

   = Arrival time of the ith vehicle - Arrival time of (i–1)st vehicle in lane j (i=1,2,...,m; j=1,2,...,N).

6. *Vehicle waiting time* = Vehicle departure time - vehicle arrival time

7. $W_{i,j}$ = Waiting time of the ith vehicle in lane j (i=1,2,...,m; j=1,2,...,N)

8. WT_j = Waiting time of all vehicles in lane j (j=1,2,...,N)

$$= \sum_{i=1}^{m} W_{i,j}$$

9. $E(W_j)$= Expected waiting time of vehicles in lane j (j=1,2,...,N).

$$= \frac{1}{m} \sum_{i=1}^{m} W_{i,j} = \frac{1}{m} WT_j$$

B. **Assumptions Regarding Objectives**

1. There is no need to consider pedestrians, bicycles, and broken down vehicles due to their negligible effect on the TI's performance.

2. Light timing sequence is the root cause of excessive vehicle waiting times.

II. Modeling Environment

A. Modeling Effort

CM outline created to illustrate the CM and to describe the traffic intersection model.

1. **Organization creating model, dates, etc.**

   Created by J. Derrick in December 1987. Revised in April 1988.

2. **Scope of the effort in time and money**

   Completed in two man-weeks.

B. Model Assumptions

1. **Boundaries**

   The system is bounded by the white intersection lines at the traffic intersection. Vehicles enter the system model by crossing the "entry" white line if

there is no queue at the corresponding light or by joining an existing lane queue. Vehicles exit the system when departing the last block in the lane.

2. Interaction with Environment

a. Input Description

(1). Vehicle Interarrival Times

The vehicle interarrival times ($IAT_{i,j}$) at each lane follow distributions which specify the input and which were determined from observation and data analysis using UNIFIT— a distribution fitting software package— by CS4150 Modeling and Simulation (Winter Quarter 1987) students. The fitted distributions (by lane designation) follow.

- Joint (Lanes 1 and 2) — did not fit a known probability distribution. Inverse transformation technique is used to generate random interarrival times from the cumulative distribution function in Appendix A.
- Lane 3 — *GAMMA* distribution

  Location parameter = 0; Scale parameter = 51.248

  Shape parameter = 1.25989; Mean = 64.5667

  Variance = 3419.5
- Lane 4 — *WEIBULL* distribution

  Location parameter = -0.01282; Scale parameter = 10.6646

  Shape parameter = 0.82821; Mean = 11.7417

  Variance = 192.766
- Lane 5 — same as Joint

- Lane 6 — *EXPONENTIAL* distribution

  Mean = 54.6774 seconds

- Lane 7 — *WEIBULL* distribution

  Location parameter = 0; Scale parameter = 34.7083

  Shape parameter = 0.86424; Mean = 37.3563

  Variance = 1692.63

- Lane 8 — *WEIBULL* distribution

  Location parameter = 0; Scale parameter = 56.0592

  Shape parameter = 0.63923; Mean = 36.8298

  Variance = 1756.41

- Lane 9 — same as Joint
- Lane 10 — same as Joint
- Lane 11 — same as Joint

(2). **Lane Travel Times**

Vehicle travel times (through the entire lane) were found to be uniformly distributed for each lane but are held constant to the mean in the model. Depending on the number of blocks and their sizes, travel times are specified by using appropriate ratios to the mean of each travel time distribution. For the purposes of the model, these travel times are attributed to the blocks as block processing times. Note that right turn on red is modeled.

- Lane 1

Observed Average Travel Time = 5.933 seconds

Designated Travel Path	=	I	Y	8	4	O	
Block Size Factor (13.1)	=	3.3	3.4	1.4	2.5	2.5	
Block Travel Time (ms)	=	1495	1540	634	1132	1132	

- Lane 2 (Straight)

Observed Average Travel Time = 4.873 seconds

Designated Travel Path	=	J	Z	9	6	3	N	D
Block Size Factor (14.6)	=	2.8	2.8	1.4	1.9	1.4	2.1	2.2
Block Travel Time (ms)	=	935	935	467	634	467	701	734

- Lane 2 (Right)

Observed Average Travel Time = 2.600 seconds

Designated Travel Path	=	J	M
Block Size Factor (5.6)	=	2.8	2.8
Block Travel Time (ms)	=	1300	1300

- Lane 3

Observed Average Travel Time = 5.036 seconds

Designated Travel Path	=	G	Q	6	8	X
Block Size Factor (12.8)	=	2.9	2.9	1.4	2	3.6
Block Travel Time (ms)	=	1141	1141	551	787	1416

- Lane 4

Observed Average Travel Time = 4.594 seconds

Designated Travel Path	=	F	P	3	2	1	O
Block Size Factor (12.4)	=	2.9	2.8	1.4	1.4	1.4	2.5
Block Travel Time (ms)	=	1074	1037	519	519	519	926

- Lane 5 (Straight)

  Observed Average Travel Time = 3.700 seconds

Designated Travel Path	=	E	N	H	L	K
Block Size Factor (11.5)	=	2.9	2.9	1.4	1.4	2.9
Block Travel Time (ms)	=	933	933	450	450	933

- Lane 5 (Right)

  Observed Average Travel Time = 3.155 seconds

Designated Travel Path	=	E	D
Block Size Factor (6)	=	3	3
Block Travel Time (ms)	=	1578	1577

- Lane 6

  Observed Average Travel Time = 6.386 seconds

Designated Travel Path	=	C	H	2	5	9	V
Block Size Factor (14.1)	=	3.3	3	1.4	1.9	1.6	2.9
Block Travel Time (ms)	=	1495	1359	634	860	725	1313

- Lane 7

  Observed Average Travel Time = 4.852 seconds

Designated Travel Path	=	B	L	1	4	7	X

Block Size Factor (14.5)	=	3.2	3	1.4	1.9	1.4	3.6
Block Travel Time (ms)	=	1071	1004	468	636	468	1205

- Lane 8

  Observed Average Travel Time = 3.660 seconds

Designated Travel Path	=	A	K				
Block Size Factor (5.1)	=	3	2.1				
Block Travel Time (ms)	=	2153	1507				

- Lane 9

  Observed Average Travel Time = 5.433 seconds

Designated Travel Path	=	S	R	4	5	3	N	D
Block Size Factor (13.8)	=	2.7	2	1.4	1.7	1.7	2.1	2.2
Block Travel Time (ms)	=	1063	788	551	669	669	827	866

- Lane 10

  Observed Average Travel Time = 3.567 seconds

Designated Travel Path	=	T	U	7	8	9	V
Block Size Factor (11.6)	=	2.7	1.7	1.5	1.4	1.4	2.9
Block Travel Time (ms)	=	830	523	461	431	431	891

- Lane 11 (Straight)

  Observed Average Travel Time = 3.967 seconds

Designated Travel Path	=	W	X	Y	Z	M
Block Size Factor (10.5)	=	3.2	1.5	1.4	1.4	3

Block Travel Time (ms) = 1209 567 529 529 1133

- Lane 11 (Right)

  Observed Average Travel Time = 2.733 seconds

Designated Travel Path = W X

Block Size Factor (5.4) = 3.4 2

Block Travel Time (ms) = 1721 1012

(3). **Selection Probabilities**

With a probability of 0.396 (123/311), a vehicle goes into lane 1.

With a probability of 0.604 (188/311), a vehicle goes into lane 2.

With a probability of 0.213 (40/188), a vehicle in lane 2 turns right.

With a probability of 0.787 (148/188), a vehicle in lane 2 goes straight.

With a probability of 0.191 (45/235), a vehicle in lane 5 turns right.

With a probability of 0.809 (190/235), a vehicle in lane 5 goes straight.

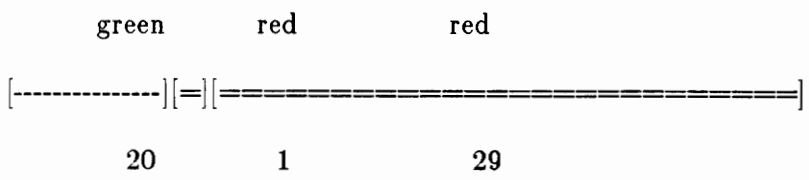
With a probability of 0.124 (33/266), a vehicle in lane 11 turns right.

With a probability of 0.876 (233/266), a vehicle in lane 11 goes straight.

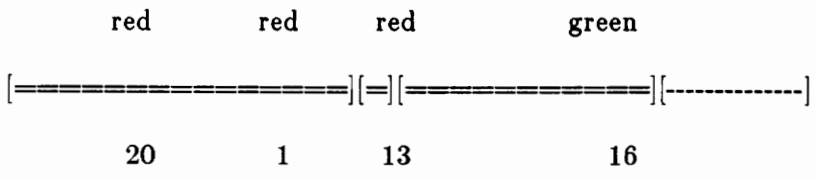
(4). **Current Light Timing Sequences**

The following light timing sequence is determined from observed data.

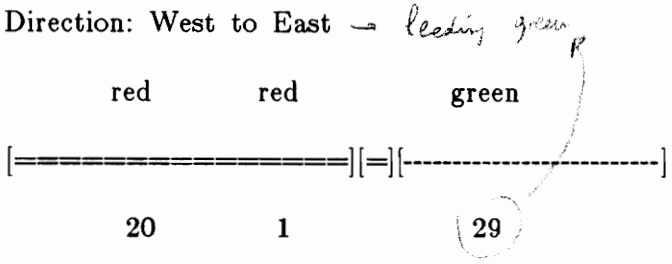
Direction: North to South and South to North



Direction: East to West



Direction: West to East → leading green



(5). **Lane Capacity Requirements**

Only 5 vehicles can wait in Lane 1 and only 5 vehicles can wait in lane 2. This means that vehicles (which have arrived in the joint lane) must wait at the head of the joint lane for available lane space in lane 1 or 2 as appropriate.

(6). **Intersection Clearance Checks**

The model simulates an intersection clearance check for all vehicles turning left into oncoming traffic (lanes 1, 3, 6, and 9). In addition, an intersection clearance check is made prior to entering the intersection. This

check is made beyond the normal checks for vehicles waiting to enter the intersection (i.e., light green [or red if turning right], and first block free). Upon a change of light from green to red, this intersection check is only made by the first vehicle in lanes 1, 2, 6, 7, 8, 9, 10, or 11 that enters the intersection at that time. Vehicles in lanes 3, 4, and 5 do not make this intersection clearance check on a light change to green since west to east traffic (with a leading green) has already checked the intersection clear. All vehicles turning right on red must check the intersection clear before proceeding. The following clearance check requirements were followed:

(a). **Left Turn Clearances**

A vehicle in lane 1 (upon reaching block 8) must check that blocks 4, 1, L, and B are empty before turning left.

A vehicle in lane 3 (upon reaching block 6) must check that blocks T, U, 7, 8, W, X, and Y are empty before turning left.

A vehicle in lane 6 (upon reaching block 5) must check that blocks 9 and Z are empty and that block J cannot be occupied by a lane 2 vehicle going straight before turning left.

A vehicle in lane 9 (upon reaching block 5) must check that blocks 2, 3, P, F, N, and E are empty before turning left.

(b). **Intersection Entry Clearances**

The first vehicle to enter the intersection from lanes 1, 2, 6, 7, or 8 after the light has just turned green must check that blocks L, H, N, 1, 2, 3, P, F, S, R, 4, 5, 6, Q, G, T, U, 7, 8, 9, Y, and Z are empty AND block

E is not occupied by a lane 5 vehicle before entering the intersection.

The first vehicle to enter the intersection from lanes 9, 10, or 11 after the light has just turned green must check that blocks B, L, 1, 4, 7, C, H, 2, 5, 8, Y, I, 6, 9, and Z are empty AND that block J is not occupied by a lane 2 vehicle going straight.

(c). **Right Turn Clearances**

A lane 2 vehicle wishing to turn right on red must check that blocks J, Z, and Y are empty AND that blocks X and W are not occupied by a lane 11 vehicle going straight.

A lane 5 vehicle wishing to turn right on red must check that blocks E, N, 3, 6, and Z are empty and that blocks 9 and J are not occupied by a lane 2 vehicle going straight.

A lane 8 vehicle wishing to turn right on red must check that blocks A, K, L, H, and N are empty and that block E is not occupied by a lane 5 vehicle going straight.

A lane 11 vehicle wishing to turn right on red must check that blocks W, B, L, 1, and 7 are empty and that block 4 is not occupied by a lane 7 vehicle going straight.

b. **Assumptions on Model/Environment Feedback or Effects**

(1) The system boundary does not include adjacent intersections. Cross effects may be observed in the model due to potential interaction during the "rush-hour" period.

(2) Random number generator seeds are strictly controlled during each replication to ensure that comparison of results using alternative light sequences will be valid.

c. **Output and Format Decisions**

The performance measures (Average Waiting Time in Lane j) with designating titles are given as output. In addition, the model constructs confidence intervals (of these performance measures) which are also provided.

3. **Initial State Definition**

The initial state definition includes:

No vehicles in system.

Light colors are north-south (green), east (red), and west (red).

4. **Simulation Termination Conditions**

The simulation is effectively terminated for each run based on the number of replications desired and the number of vehicles to be processed. For a given light sequence, the model conducts a certain number (say x) of replications. Each replication warms up the system by processing a certain number of vehicles (say y) which specifies a transient period. Following the transient period, the steady state period for each replication is based on the processing of an additional number of vehicles (say z). The values of x, y, and z are prescribed by the input.

III. Model Definition

A. Model

1. Sets: None

2. Indicative Attributes

System time: temporal transitional indicative

Units: 1/1000 seconds

Maximum number of departures from system in transient period: permanent indicative, with value assigned at system time = zero.

Maximum number of departures from system in steady-state: permanent indicative, with value assigned at system time = zero.

Number of departures from system in transient period: temporal transitional indicative

Number of departures from system in steady-state: temporal transitional indicative

Units (all departures): Integer, Number of vehicles

Number of model replications: permanent indicative, with value assigned at system time = zero.

Number of Lanes: permanent indicative, with value assigned at system time = zero.

Units: Integer

3. Relational Attributes: None

B. Submodels

1. Vehicle Submodel

a. Sets

Set of vehicles (dset) served in lane j (j=1,2,...,N)

b. Indicative Attributes

Waiting time of vehicles in lane j, WT_j (j=1,2,...,N): temporal transitional indicative

Units: seconds

Total waiting time of vehicles in lane j (j=1,2,...,N): permanent indicative, with value = WT_j assigned when *number of departures from system in steady-state = Maximum number of departures from system in steady-state.*

Units: seconds

Number of vehicles departing from lane j, m (j=1,2,...,N): temporal transitional indicative

Total number of vehicles departing from lane j (j=1,2,...,N): permanent indicative, with value = m assigned when *number of departures from system in steady-state = Maximum number of departures from system in steady-state.*

Expected waiting time of vehicles in lane j, $E(W_{i,j})$ (j=1,2,...,N): permanent indicative, with value assigned when *number of departures from system in steady-state = Maximum number of departures from system in steady-state.*

Units: seconds

c. Relational Attributes: A reference mechanism is necessary for the set of vehicles served in lane j (j=1,2,...,N).

(1). Base level of vehicle submodel: Each vehicle is an object.

(a). Sets: members of the set of vehicles served in lane j (j=1,2,...,N).

(b). Indicative Attributes

Waiting time: permanent indicative, with value assigned at time of vehicle's departure from system.

Arrival time: permanent indicative, with value assigned at time of vehicle's arrival to system.

Direction of movement (right or straight): permanent indicative, with value assigned at vehicle's arrival to system.

Lane identifier (Lane number): permanent indicative, with value assigned at vehicle's arrival to system.

(c). Relational Attributes: Each vehicle has membership in the set of vehicles served in lane j (j=1,2,...,N).

2. **Light Submodel** (Base Level): The light is an object.

a. Sets: None

b. Indicative Attributes

Sequence duration (for a particular color): permanent indicative, with value assigned at system time = zero.

Units: seconds

Direction (North-south, West, East): permanent indicative, with value assigned at system time = zero.

Direction color (red, green): status transitional indicative

c. Relational Attributes: None

3. **Intersection Submodel**

a. Sets: A reference mechanism is necessary for the set of blocks which form a path through the intersection for the vehicle in lane j (j=1,2,...,N).

b. Indicative Attributes

Clear for left turn: status transitional indicative

Clear for entry: status transitional indicative

Clear for right turn: status transitional indicative

Number of Blocks: permanent indicative, with value assigned at system time = zero.

c. Relational Attributes: None

(3). **Block** (Base Level): The block is an object.

(a). Sets

Set of blocks (dset) which define a vehicle path from lane j (j=1,2,...,N)

(b). Indicative Attributes

Processing time for vehicles which transit the block: permanent indicative, with value assigned at system time = zero.

Units: seconds

Identifier: permanent indicative, with value assigned at system time = zero.

Status (busy, idle): status transitional indicative

In use by (Transiting Vehicle identifier): status transitional indicative

(c). Relational Attributes

Each block is a member in the set of blocks which define a vehicle path from lane j (j=1,2,...,N)

3.2 The ES CF Application

The SIMSCRIPT [Kiviat et al. 1983; CACI 1983] Simulation Programming Language (SPL) is used to demonstrate the ES CF. Although the later versions of SIMSCRIPT allow the development of models based on the PI CF, such features are not included and thus a pure ES CF model is obtained. Since SIMSCRIPT is an SPL, it provides ease of event manipulation of the events list, (including stochastic scheduling by common probability distributions) and built-in statistical collection methods. This section includes a detailed discussion of

- the SIMSCRIPT preamble in which the appropriate declarations of model program variables and components are given,
- the Event Routines, the most singularly distinguishing features of an ES CF,

and a brief discussion of

- the SIMSCRIPT main routine, the controlling executive of this ES CF model, and
- the statistics output of the model.

3.2.1 The Preamble

The **preamble**, portions of which are shown in Figure 3.2, gives a clear indication of some of these above mentioned features. The event notices (for those event routines which are provided as coded procedures in the body of the program) are declared in the preamble. Event notice declarations may include the specification of any arguments that will be

```
preamble
1  event notices include
2     turn.ns.red and turn.ns.green and turn.west.green
3     and turn.east.green                      ''Event arguments include:
4     every departure has a out.vehicle        '' outgoing car
5     every arrival.blockd has a moving.car.d '' incoming car to block
6     every arrival.blockh has a moving.car.h
7     ...   ...   ...   ...   ...
8     every arrival.blockz has a moving.car.z
9     every arrival.block1 has a moving.car.1
10    ...   ...   ...   ...   ...
11    every arrival.block9 has a moving.car.9
12    every turning.left has a left.moving.car ''Car making turn
13    every enter has a in.vehicle              ''Car entering intersection
14    every arrival.joint has a incoming.car12 ''Car arriving lane
15    every arrival.lane1 has a incoming.car1
16    ...   ...   ...   ...   ...
17    every arrival.lane11 has a incoming.car11

18 normally, mode is integer
19 permanent entities
20    every light has a ns.color, a west.color and a east.color
21    every block has a status, a laneuser, a turner and owns a block.queue
22    every lane owns a lane.queue
23    ...   ...   ...   ...   ...
24 temporary entities
25    every car has an arrtime, a laneid,an id and a to.right
       and may belong to a block.queue
       and may belong to a lane.queue
       and may belong to a turn.queue
26    define arrtime as a real variable

27 The system has a deps.in.ss,              ''departures in steady state
28    a length.of.tp, a length.of.ss,       ''lengths: transient, ss pds.
29    a deps.fm.1, a deps.fm.2, a deps.fm.2r,''departure variables/lane
30    ...   ...   ...   ...   ...
31    a wait.in.1, a wait.in.2, a wait.in.2r,''waiting times/lane
32    ...   ...   ...   ...   ...
33    a numrng, a numruns,                  ''num of rn generators, runs
34    a clearedns, a clearedwe,             ''boolean, intersection clear
35    a counter, a debug, an i,             ''utility variables
36    a gen.1.2 random linear variable,     ''variables for inverse
37    a gen.5 random linear variable,       '' transformation variate gen
38    ...   ...   ...   ...   ...
39    define wait.in.1 as a real variable   ''type definitions
40    define wait.in.2 as a real variable   '' and various index assign-
41    define wait.in.2r as a real variable  '' ments.
42    ...   ...   ...   ...   ...
43    define expon as real fortran function
44    ...   ...   ...   ...   ...
45    tally mean.wait.in.1 as the mean of wait.in.1
46    tally mean.wait.in.2 as the mean of wait.in.2
47    ...   ...   ...   ...   ...
48 end
```

Figure 3.2 Portions of SIMSCRIPT Preamble from ES CF Application

required for the event routines (e.g., **out.vehicle** as an argument to event **departure** in line 4). Notice declarations are included for event routines, for example, that represent the changing of the color of the light (e.g., **turn.ns.red**), departures from the intersection (**departure**), arrivals to the intersection blocks (e.g., **arrival.blockd**), entrance to the intersection transit area (**enter**), and arrivals to the lanes (e.g., **arrival.lane1**). The preamble identifies the **permanent entities** of the model (e.g., the light, the blocks, and the lanes, in lines 19-22) and the **temporary entities** (the vehicles or cars, line 25). Useful attributes are also declared for these model entities or objects. External functions are identified in the preamble. An example of this, shown in line 43, is **expon**, an external Fortran function. The **tally** key words (lines 45,46) provide the ease of calculating the mean waiting times of vehicles in each lane for this model.

3.2.2 The Event Routines

In this section, we discuss the principal event routines which are included in the ES CF application to the TI,

- Light color changes,
- Lane arrivals,
- Vehicle departure,
- Vehicle entrance to the intersection, and
- Block arrivals.

In accordance with the discussion in Section 2.3, the events associated with the changing of the light color (as well as the lane arrival and departure events) are determined events. The events which consider vehicle entrance to the transit area and arrival to the indivi-

dual blocks are contingent events in that they depend on conditions of light color (in the case of the "enter" event), etc., and on the availability of the block for an arrival (i.e., Is there a vehicle already in that particular block?).

Figure 3.3 shows the **turn.ns.green** event routine which represents the state change of the north-south direction of the light to the green color. As mentioned, this is a determined event in that the timing of the light color changes can be predetermined. Note that on line 12 this event also includes the bootstrapping of the next color change, accomplished by the use of the SIMSCRIPT **schedule** primitive. Use of the "schedule" primitive places an event notice on the event list at its appropriate ordered position. Here, the state change of the north-south direction of the light to red (by the "turn.ns.red" event routine) is scheduled in 20 seconds.

In general, the "turn.ns.green" routine includes statements which demonstrate how vehicle movement into the intersection transit area is managed, but more importantly, it clearly shows the complexity of interaction among event routines that is discussed in Section 2.3. Pidd [1984] states that the event routine must identify all possible actions that can occur as a result of the state change associated with the event. The identification of contingent events which result from the state change makes the programming task difficult for the modeler, especially when the system being modeled is as complex as the TI.

The state change of the "turn.ns.green" event is accomplished by the assignment statements in lines 1,2 and 3. Note also that line 4 sets a control variable, **clearedwe**, to false. This indicates that the intersection must be re-cleared for vehicles to enter the intersection from east-west directions. (See Section 3.1 for a full description of the conditions that are necessary for determining entrance to the intersection.) Lines 5 through 11

```
''*********************************************************************
''DESCRIPTION: Event TURN.NS.GREEN; Sets ns, east, and west light
''      direction colors;  schedules turn.ns.red during run, otherwise
''      sets ns color red blocking intersection with all red.  This
''      will clear intersection, prevent further entries, and end run.
''      Also sets west-east clearance flag to false.
''
''ATTRIBUTES: None
''
''INPUT(S) : None
''
''OUTPUT(S): Light attributes of color are set.
''
''CALLS     : test.entry for all north-south lanes, and east-west right
''       right turning lanes.
''CALLED BY: None, scheduled by turn.east.green.
''*********************************************************************;
event turn.ns.green
1       let ns.color(1) = green              ''Set color attributes
2       let west.color(1) = red
3       let east.color(1) = red
4       let clearedwe = false                ''Set clearance flag to False
5       call test.entry.9.to.11(11,lane11,block.w)
6       call test.entry.345(5,lane5,block.e)
7       call test.entry.12(1,lane1,block.i) ''Test various entries
8       call test.entry.12(2,lane2,block.j)
9       call test.entry.678(6,lane6,block.c)
10      call test.entry.678(7,lane7,block.b)
11      call test.entry.678(8,lane8,block.a)
12      if deps.in.ss <= length.of.ss        ''Schedule next light change
           schedule a turn.ns.red in 20 seconds
13      else
           let ns.color(1) = red             ''Block intersection
14      always
15      end
```

Figure 3.3 Event TURN.NS.GREEN

are calls to user-defined routines (**test.entry**) that identify the possible actions that might follow from the state change. Figure 3.4 is one such "test.entry" routine that determines if vehicles can enter from lanes 6, 7, or 8. If conditions for entry are met, then the "enter" event routine, a contingent event, is immediately scheduled. Checking all actions that can occur following a state change can, therefore, be a quite tedious and error-prone task.

Figures 3.5 and 3.6 give examples of other determined event routines, a lane arrival routine and the departure routine. The **arrival.lane1** event routine of Figure 3.5 demonstrates the scheduling of future arrivals and the prevalent complexity of interactions among events. Conditions may permit an entrance to the intersection immediately after arrival to lane 1. In any case, a new arrival can be scheduled. The **departure** event routine, Figure 3.6, includes the calculation of important statistical information (e.g., lane waiting times) and frees the last block that was transited by the departing vehicle with the user-defined **releese** routine (e.g., lines 25, 28, or 30 of Figure 3.6).

Contingent events **enter** and a block arrival routine, **arrival.blockd**, are shown in Figures 3.7 and 3.8. These events in turn may result in other instances of contingent or determined events. The issue of complexity in interaction is again demonstrated.

3.2.3 The Simulation Executive or Main

Figure 3.9, the SIMSCRIPT **main** routine, is the executive that controls the execution of the simulation. The initialization of variables, seed values and associated matrices, and the input of variate generation data and the creation model objects are accomplished. The method of replications is performed using a looping construct (line 12), looping on the number of desired runs or replications. The initial vehicle arrivals are generated

```
''*************************************************************************
''DESCRIPTION: Routine TEST.ENTRY.678; Checks conditions for entry from
''      Lanes 6,7,and 8.
''
''ATTRIBUTES: None
''
''INPUT(S)   : Lane and queue id's of waiting cars,
''      and first block into intersection from their lane
'       are passed as arguments.
''
''OUTPUT(S) : When conditions are right, entry is scheduled.
''
''CALLS      : None
''
''CALLED BY : Turn.ns.red, turn.ns.green, various arrival.block routines.
''*************************************************************************;
routine test.entry.678(lane.num,queue.id,first.block)
1     select case lane.num
2     case 6,7
3        if lane.queue(queue.id) is not empty then
          ''There are cars waiting to enter from lane.num
4        if ((ns.color(tfclight) = green) and (status(first.block) = idle)
          and ((intersection(ns) = idle) or (clearedns = true)))
          ''Conditions satisfied, remove car at head of waiting queue
          ''and schedule that car for an enter event.
5           remove the first waiting.car from lane.queue(queue.id)
6           schedule an enter given waiting.car now
7        always
8     case 8
9        if lane.queue(queue.id) is not empty then
          ''There are cars waiting to enter from lane 8 that may enter
          ''a green light or on a red (right turn) if conditions are right.
10       if (((ns.color(tfclight) = green) and (status(first.block) = idle)
          and ((intersection(ns) = idle) or (clearedns = true))) or
          ((ns.color(tfclight) = red) and
          (to.right(f.lane.queue(queue.id)) = true) and
          (status(first.block) = idle) and (right.ok(queue.id) = true)))
          ''Conditions satisfied, remove car at head of lane 8 queue
          ''and schedule that car for an enter event.
11          remove first waiting.car from lane.queue(queue.id)
12          schedule an enter given waiting.car now
13       always
14    default
15    endselect
16    return
17    end
```

Figure 3.4 User-defined Routine TEST.ENTRY.678

```
''*************************************************************************
''DESCRIPTION: Event ARRIVAL.LANE1; Car is moved as far as it can go...
''     into lane1 or on into intersection.  Also a check is included
''     for cars waiting in the joint lane to see if any of these cars
''     can be moved into lane1 or 2 (within capacity of 5 limitations).
''
''INPUT(S) : Incoming car, arriving to lane 1
''
''OUTPUT(S): Car is appropriately transferred to next destination.
''
''CALLS     : None
''
''CALLED BY: Test.entry.12 and self
''*************************************************************************;
event arrival.lane1 given car1
1     if lane.queue(lane1) is not empty       ''Must stop in lane 1
2        file car1 in lane.queue(lane1)
3     else                                    ''Check for entry to inters.
4        if ((ns.color(tfclight) = red) or (status(block.i) = busy) or
            ((intersection(ns) = busy) and (clearedns = false)))
5           file car1 in lane.queue(lane1)   ''Entry not possible
6        else
7           schedule an enter given car1 now ''OK, enter intersection
8        always
9     always
10    if lane.queue(joint) is not empty       ''Check for cars in joint
11       if ((laneid(f.lane.queue(joint)) = 1) and        ''Find lane1 car
            (n.lane.queue(lane1) < 5))
12          remove first joint.car from lane.queue(joint)
13          schedule an arrival.lane1 given joint.car now
14       else
15          if ((laneid(f.lane.queue(joint)) = 2) and     ''Find lane2 car
               (n.lane.queue(lane2) < 5))
16             remove first joint.car from lane.queue(joint)
17             schedule an arrival.lane2 given joint.car now
18          always
19       always
20     always
21  end
```

Figure 3.5 Event ARRIVAL.LANE1

```
''*************************************************************************
''DESCRIPTION: Event DEPARTURE; Depending on the transient and steady-
''     state durations, and on the laneid of the departing car,
''     the waiting time of the car is determined and accumulated into
''     the overall waiting time for that lane.  Also in some cases, the
''     last block in the departing car's path is released.
''
''INPUT(S) : The departing car is passed to the event.
''
''OUTPUT(S): Performance measure variables are updated, some blocks are
''      released.
''CALLS     : releese
''
''CALLED BY: None, scheduled by arrival.block events (end path blocks).
''*************************************************************************;
event departure given a.car
1    if length.of.tp > 1                    ''Decrement transient pd
2       length.of.tp = length.of.tp - 1     '' counter, nothing else
3    else
4      if length.of.tp = 1                 ''Decrement tp once more
5         length.of.tp = 0
6      else                                ''Now into steady-state pd.
7         deps.in.ss = deps.in.ss + 1      ''Keep track of deps in ss
8         if deps.in.ss <= length.of.ss    ''Calculate waiting times if
9            select case laneid(a.car)     '' during steady-state pd.
10            case 1
11               wait.in.1 = time.v - arrtime(a.car)
12            case 2
13               if to.right(a.car) = true
14                  wait.in.2r = time.v - arrtime(a.car)
15               else
16                  wait.in.2 = time.v - arrtime(a.car)
17               always
                 ...   ...   ...   ...   ...
18            default
19            endselect
20         always
21       always
22    always
23    select case laneid(a.car)
24    case 1
25      call releese(block.o)
26    case 2
27      if to.right(a.car) = true
28         call releese(block.m)
29      else
30         call releese(block.d)
31      always
      ...   ...   ...   ...   ...
32    default
33    endselect
34    destroy the car called a.car
35    end
```

Figure 3.6 Portions of Event DEPARTURE

```
''*************************************************************************
''DESCRIPTION: Event ENTER; Depending upon where the intersection is
''     entered, the first block is setbusy and an appropriate arrival
''     to the next block is scheduled.
''
''INPUT(S) :  The entering car is passed to the event.
''
''OUTPUT(S):  The car is moved into the first block of the intersection
''     and the next block arrival for the car is scheduled.
''CALLS     :  setbusy
''
''CALLED BY:  various test.entry routines, various arrival.joint/lane
''       events.
''*************************************************************************;
event enter given a.car
1    select case laneid(a.car)
2    case 1
3       call setbusy(block.i, a.car)
4       schedule an arrival.blocky given a.car in 1.495 seconds
5    case 2
6       call setbusy(block.j, a.car)
7       if to.right(a.car) = true
8          schedule an arrival.blockm given a.car in 1.300 seconds
9       else
10         schedule an arrival.blockz given a.car in 0.935 seconds
11      always
12   case 3
13      call setbusy(block.g, a.car)
14      schedule an arrival.blockq given a.car in 1.141 seconds
15   ...    ...    ...    ...    ...
16   case 8
17      call setbusy(block.a, a.car)
18      schedule an arrival.blockk given a.car in 2.153 seconds
19   case 9
20      call setbusy(block.s, a.car)
21      schedule an arrival.blockr given a.car in 1.063 seconds
22   case 10
23      call setbusy(block.t, a.car)
24      schedule an arrival.blocku given a.car in 0.830 seconds
25   case 11
26      call setbusy(block.w, a.car)
27      if to.right(a.car) = true
28         schedule an arrival.blockx given a.car in 1.721 seconds
29      else
30         schedule an arrival.blockx given a.car in 1.209 seconds
31      always
32   default
33   endselect
34 end
```

Figure 3.7 Portions of Event ENTER

```
''*********************************************************************
''DESCRIPTION: Event ARRIVAL.BLOCKD; Checks block for availability;
''      If not available, put car in block queue, else...
''      set block busy, release appropriate blocks or setidle;
''      If event is for arrival to first block in path, do appropriate
''        test.entry check; if last block, schedule departure;
''      If block impacts any clearance check (intersection entry, right
''        turn, or left turn) do appropriate test.entry or test.left.
''
''INPUT(S) : Arriving car is passed to event.
''
''OUTPUT(S): As indicated in above description.
''
''CALLS     : Various test.entry, test.left, turning.left routines
''         (as applicable), as well as setbusy, setidle, releese.
''CALLED BY: For first blocks, sched. by enter event;  If block after
''        a turn, sched. by turning.left event; Typically scheduled by
''        the arrival.block event for the block preceding it in the path
''*********************************************************************;
event arrival.blockd given a.car
1     if status(block.d) = busy                ''Block is busy, queue up
2        file a.car in block.queue(block.d)
3     else                                     ''Block available so
4        call setbusy(block.d,a.car)           '' set block busy
5        select case laneid(a.car)
6        case 2                                ''Lane 2 car
7           call releese(block.n)              '' free block n, test.entry,
                                               '' and sched departure.
8           call test.entry.345(5,lane5,block.e)
9           schedule a departure given a.car in 0.734 seconds
10       case 5                                ''Lane 5 car
11          call setidle(block.e)              '' free block e, test.left
                                               '' for lane9, test.entry, and
                                               '' sched a departure
12          call test.left(from.9, lane9, block.3)
13          call test.entry.345(5,lane5,block.e)
14          schedule a departure given a.car in 1.577 seconds
15       case 9                                ''Lane 9 car
16          call releese(block.n)              '' free block n, test.left
                                               '' for lane9, test.entry, and
                                               '' sched a departure
17          call test.left(from.9,lane9,block.3)
18          call test.entry.12(1,lane1,block.i)
19          call test.entry.12(2,lane2,block.j)
20          call test.entry.678(6,lane6,block.c)
21          call test.entry.678(7,lane7,block.b)
22          call test.entry.678(8,lane8,block.a)
23          schedule a departure given a.car in 0.866 seconds
24       default
25       endselect
26    always
27  end
```

Figure 3.8 Event ARRIVAL.BLOCKD

```
''*************************************************************************
''DESCRIPTION: Main; Performs initializations, schedules initial
''     arrivals, starts the simulation, tallies statistics.
''
''ATTRIBUTES: None
''
''INPUT(S) : Cumulative distribution data, lanes 1,2,5,9,10,11.
''
''OUTPUT(S): Statistical data, average waiting times per lane.
''
''CALLS    : buildseeds, load.seeds, make.objects, statistics,
''    init.run, setup.next.run, load.stats
''CALLED BY: None
''*************************************************************************;
main
1    call init.run
2    call build.seeds(numrng,numruns) yielding seed.matrix(*,*)
3    release seed.v(*)                          ''Setup seeds for use
4    call load.seeds(numrng,1)  yielding seed.v(*)
5    reserve lane1.matrix(*), lane2.matrix(*), lane2r.matrix(*),
6       lane3.matrix(*), lane4.matrix(*), lane5.matrix(*),
7       ...    ...    ...    ...    ...
8    read gen.1.2                               ''Read input data
9    ...    ...    ...    ...    ...
10   call make.objects
11   call set.output
12   for i = 1 to numruns                       ''Loop for replications
13      do
14      create a car called car12               ''Create initial car for ea
15      create a car called car3                '' lane.
16      ...    ...    ...    ...    ...
17      schedule a turn.ns.green now            ''Start light sequence
                                                ''Sched initial arrivals
18      schedule an arrival.joint given car12 in gen.1.2 seconds
19      schedule an arrival.lane3 given car3 in gamma.f(64.5667,1.25989,5) seconds
20      schedule an arrival.lane4 given car4 in weibull.f(0.82821,10.6646,6) seconds
21      schedule an arrival.lane5 given car5 in gen.5 seconds
22      schedule an arrival.lane6 given car6 in expon(54.6774,seed.v(9)) seconds
23      schedule an arrival.lane7 given car7 in weibull.f(0.86424,34.7083,10) seconds
24      schedule an arrival.lane8 given car8 in weibull.f(0.63923,56.0592,11) seconds
25      schedule an arrival.lane9 given car9 in gen.9 seconds
26      schedule an arrival.lane10 given car10 in gen.10 seconds
27      schedule an arrival.lane11 given car11 in gen.11 seconds
28      start simulation
29      call load.stats(i)                      ''Load perf. measure arrays
30      call setup.next.run
31      if i < numruns                          ''Load seeds for next run
32         release seed.v(*)
33         call load.seeds(numrng, i+1) yielding seed.v(*)
34      always
35      loop
36      call statistics(numruns)                ''Output data
37   end
```

Figure 3.9 Portions of SIMSCRIPT Main Routine

using the SIMSCRIPT defined distribution functions or the user-defined functions in lines 18-27. Statistical information is included in the output of the model following the completion of all replications. The **start simulation** statement at line 28 begins the execution of event routines for which there are notices on the event list. The processing of the event list is automatically performed and continues until there are no further notices on the event list.

3.2.4 The Statistical Output

The statistical output, Figure 3.10, is called at the end of the simulation. The mean waiting time of all vehicles that have departed a particular lane have been recorded in matrices where the name of the matrix indicates the lane of interest. In this way, the performance measures of the simulation study are produced. Figure 3.11 shows an example of the output(s) from a model execution of three replications.

3.3 The AS CF Application

Activity Cycle Diagrams (ACDs) which are discussed below are used to introduce the application of the AS CF to the TI. Due to the complexity of the TI with its large number of cooperating entities (transiting cars and intersection blocks, in particular), a single ACD which incorporates all these interrelations could not practically fit within the space limitations of this thesis. Therefore, selected portions of the ACD for the TI are provided which should lead one to an understanding of the complete ACD and to an appreciation of its complexity.

Following the selected ACDs, the activity descriptions which are suggested by these ACDs are given in a Pascal-like pseudocode. The scope of coverage is restricted once

```
''************************************************************************
''DESCRIPTION: Routine STATISTICS;  Displays the performance measures,
''      the average waiting times for the cars in each lane... and
''      displays the confidence interval calculations.
''
''INPUT(S) : number of replications is passed in to routine.
''
''OUTPUT(S): Performance measures of study
''
''CALLS     : None
''
''CALLED BY: main
''************************************************************************;
routine statistics(numruns)
1     define i as an integer variable
2     use 7 for output
3     for i = 1 to numruns
4        do
5        print 1 line with lane1.matrix(i), lane2.matrix(i),
            lane2r.matrix(i), lane3.matrix(i), lane4.matrix(i), and
            lane5.matrix(i) thus
  ***.****  ***.****  ***.****  ***.****  ***.****  ***.****
6        loop
7     use 8 for output
8     for i = 1 to numruns
9        do
10       print 1 line with lane5r.matrix(i), lane6.matrix(i),
            lane7.matrix(i), lane8.matrix(i), lane9.matrix(i), and
            lane10.matrix(i) thus
  ***.****  ***.****  ***.****  ***.****  ***.****  ***.****
11       loop
12    use 9 for output
13    for i = 1 to numruns
14       do
15       print 1 line with lane11.matrix(i) and lane11r.matrix(i) thus
  ***.****  ***.****
16       loop
17    use 6 for output
18    return
19    end
```

Figure 3.10 User-defined Routine STATISTICS

```
6
Average waiting time in lane 1
Average waiting time in lane 2 (str)
Average waiting time in lane 2 (rt)
Average waiting time in lane 3
Average waiting time in lane 4
Average waiting time in lane 5 (str)
 20.1405   17.6129   11.5898   20.2876   18.2098   18.2914
 20.2678   17.2750   10.9285   20.1956   18.6365   18.0423
 20.4581   17.6032   11.3787   19.8717   18.3187   18.4496

6
Average waiting time in lane 5 (rt)
Average waiting time in lane 6
Average waiting time in lane 7
Average waiting time in lane 8
Average waiting time in lane 9
Average waiting time in lane 10
 14.2450   19.8322   16.3817    5.8213   13.0041   10.0023
 13.1426   20.3903   16.2367    5.7599   13.1836    9.7661
 13.1975   19.7866   16.6054    5.8273   13.3474   10.0401

2
Average waiting time in lane 11 (str)
Average waiting time in lane 11 (rt)
 11.6745    8.3365
 11.8405    8.4285
 11.6309    8.3136
```

Figure 3.11 Output of Three Replications from SIMSCRIPT Model

again due to the large number of activity descriptions which would be necessary in a complete representation of the AS CF for the TI. Therefore, only a representative sampling of activity descriptions (as described below) are given. A more comprehensive listing (names only) of the possible activity descriptions in a complete representation is shown, however, to indicate this complexity. The activity descriptions which correspond to the "given" ACDs and which deal with Lane 1 car entities and their associated model components are covered to provide the necessary link in comprehension from the ACD to a possible implementation. The limited coverage of activity descriptions is sufficient for the purposes of this thesis.

The selected ACDs, their associated activity descriptions, and the previous discussion of the AS CF combine to demonstrate the essential concepts of the AS CF.

3.3.1 Activity Cycle Diagrams

The Activity Cycle Diagram (or ACD) has been proclaimed to be a useful tool in the representation of simulation models in a way which is understandable to managers and programmers alike. The ACD is most often associated with the AS CF. However, it has been claimed to be more generally applicable [Pidd 1984]. This brief description of the ACD is a concise summary of the discussions of Pidd [1984], Hutchinson [1975], and Mathewson [1974]. ACDs are attributed to Tocher [1966]. Related terms include *entity-cycle diagrams, wheel charts* or *wheel-cycle diagrams, entity-activity diagrams,* and *HOCUS diagrams.* O'Keefe and Davies [1987] suggest a slight variation called an *activity flow diagram.* Tocher [1963] illustrates some general simulation problems with the diagrams but the diagrams and their usage were apparently not popularized until a few years later.

According to Pidd [1984], the ACD can graphically represent the interactions of the entities of a model at a high level. When using the ACD, one must identify the entities of the model, the activities in which each participates, and how the entities and their activities relate to one another. In a sense, a graphical life history of each entity and their interactions as a whole are provided. A precise specification of the details of these interactions is often not shown, however. The diagram serves as a starting point in the design process which can be further refined (into other forms) as necessary. Hutchinson [1975] states that a "completed" ACD includes activity durations in time units, queue disciplines to be followed (FIFO, etc.), and the starting conditions of the model. In practice it seems, however, that such details of ACD representation vary from modeler to modeler. This review covers only the most widely used and basic components of the ACD.

The principal components of an ACD are the *active* and *dead* state symbols. The active and dead states could be conveniently referred to as busy and idle states and are represented by squares and circles, respectively.

An active state describes an activity of an entity in which there is cooperation with another model entity or entities. In addition, the active state may only require one entity [Hutchinson 1975]. The duration of the active state can be determined in some way, perhaps from a suitable probability distribution. A dead state is a period of unknown duration during which the entity is waiting for some condition(s) to hold. Before an entity can proceed into an active state from a dead state, all necessary cooperating entities for the upcoming active state must be available. Dead states are most often represented as queues. No activity occurs in the dead state since the entity does not experience any changes of state in this waiting condition. Mathewson [1974] discusses two

particular dead states (beyond those already mentioned) which have additional symbols. The *infinity queue* represents an infinite source or sink from which entities may be taken for entrance to the system or to which they may be returned upon exit from the system. A *flag* indicates a queue of entitities for which the queue discipline is of no interest. Thus, the ordering of entities is not maintained by a flag. Instead, the flag is simply a counter which registers the total number of entities which occupy the queue. Two dead states, touching side-by-side, so as to form the familiar infinity symbol, are used to represent the infinity queue. The flag is simply a dead state with a diagonal line inscribed within it.

By convention, the progression or flow of an entity's activities (as shown by the ACD) is an alternating sequence of dead and active states. "Dummy" queues or dead states [Hutchinson 1975] are often used in the ACD preceding an active state which requires only one entity. The dummy queue serves to maintain the convention of alternating dead and active states, but no time is spent in them by the entities which are passing through.

The ACD is able to clearly depict a system's entities and the activities in which each entity is engaged. The cooperating activities are easily identified. The ACD focuses entirely on a system's entities and is independent of system materials, the number of entities in the system, and the time requirements of the individual entity activities. The ACD provides a "sound basis for a discussion of the logic" of a system and represents a suitable foundation upon which a simulation can be built [Hutchinson 1975].

3.3.2 Identification of Model Components for the AS CF

To build the ACDs which will be used to demonstrate the AS CF application of the TI, the entities that are involved must first be identified. The following classes of model entities (quantities in parentheses) can be specified:

Imaginary, permanent entities

Arrival machines for the Joint Lane and Lanes 3 to 11 (10)

Real, permanent entities

Light (1)

Blocks A to Z (26)

Blocks 1 to 9 (9)

Real, temporary entities

Cars (numerous)

Resources

Lane 1 space (5 cars or less)

Lane 2 space (5 cars or less)

3.3.3 Listing of Possible Activities

Arrival Activities

Activity Name	**Identifier**	**Participating entities/resources**
Arrival to Joint Lane	(ARRJ)	Lane 1/2 Car, Joint Arrival Machine
Arrival to Lane 3	(ARR3)	Lane 3 Car, Lane 3 Arrival Machine
.....	.....	.....
Arrival to Lane 11	(ARR11)	Lane 11 Car, Lane 11 Arrival Machine

Light Activities

Activity Name	**Identifier**	**Participating entities/resources**
Light, NS green	(L1)	Light only
Light, NS red	(L2)	Light only
Light, West green	(L3)	Light only
Light, East green	(L4)	Light only

Finish Activities

Activity Name	**Identifier**	**Participating entities/resources**
End Transit Block A	(ETRANSA)	Block A, Lane 8 Car
End Transit Block B	(ETRANSB)	Block B, Lane 7 Car
. . . . .	. . . . .	. . . . .
End Transit Block Z	(ETRANSZ)	Block Z, Lane 2 or 11 Car
End Transit Block 1	(ETRANS1)	Block 1, Lane 4 or 7 Car
End Transit Block 2	(ETRANS2)	Block 2, Lane 4 or 6 Car
. . . . .	. . . . .	. . . . .
End Transit Block 9	(ETRANS9)	Block 9; Lane 2,6, or 10 Car

Start Activities

Activity Name	**Identifier**	**Participating entities/resources**
Begin Transit Block A	(BTRANSA)	Block A, Lane 8 Car
Begin Transit Block B	(BTRANSB)	Block B, Lane 7 Car
. . . . .	. . . . .	. . . . .
Begin Transit Block Z	(BTRANSZ)	Block Z, Lane 2 or 11 Car
Begin Transit Block 1	(BTRANS1)	Block 1, Lane 4 or 7 Car

Begin Transit Block 2	(BTRANS2)	Block 2, Lane 4 or 6 Car
.....	.....	.....
Begin Transit Block 9	(BTRANS9)	Block 9; Lane 2,6, or 10 Car

Special Activities

Activity Name	**Identifier**	**Participating entities/resources**
Split to Lane 1 or 2	(SPLIT)	Lane1/2 Car, Lane 1/2 Space Resource
Turn Left in Lane 1	(TURN.LEFT.1)	Lane 1 Car only
Turn Left in Lane 3	(TURN.LEFT.3)	Lane 3 Car only
Turn Left in Lane 6	(TURN.LEFT.6)	Lane 6 Car only
Turn Left in Lane 9	(TURN.LEFT.9)	Lane 9 Car only

3.3.4 Specific Activity Cycle Diagrams

The above identification of model entities and associated activities suggests the following limited set of ACDs. Single ACDs are given which associate with the primary entities that interact with Lane 1 Cars during its arrival, entry, and transit of the traffic intersection. Figure 3.12 shows the Light's ACD with active states in different colors. Figure 3.13 is a representative sample of Block ACDs for those blocks in a Lane 1 Car's path. Figure 3.14 depicts the ACD for a Lane 1 Car showing the various active and dead states through which the car proceeds. Finally, a coordinated ACD is given in Figure 3.15 which attempts to demonstrate the complex interactions that occur as a Lane 1 car travels through the intersection.

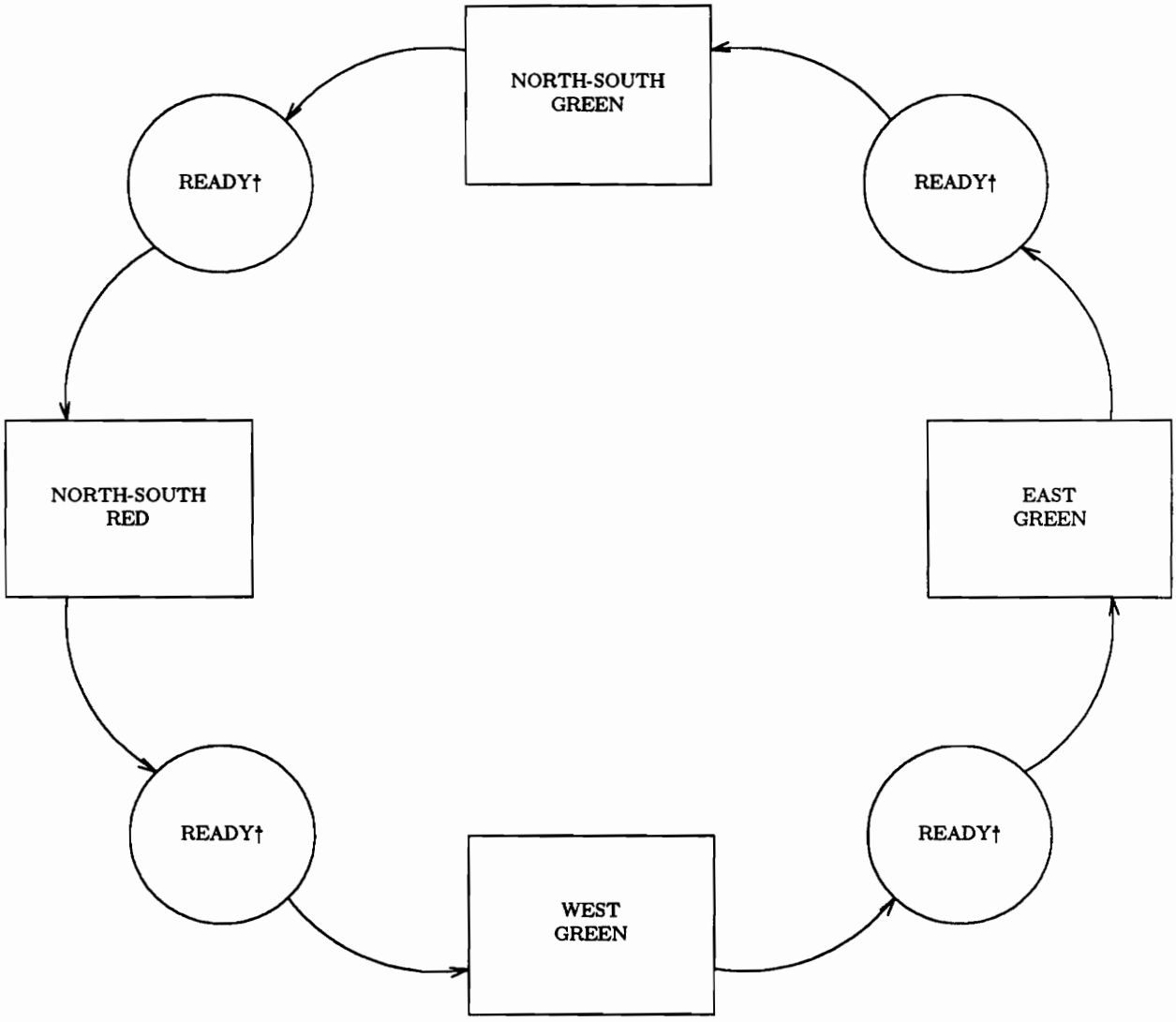


Figure 3.12 The Light Activity Cycle Diagram

† Dummy queues; No time is spent in them.

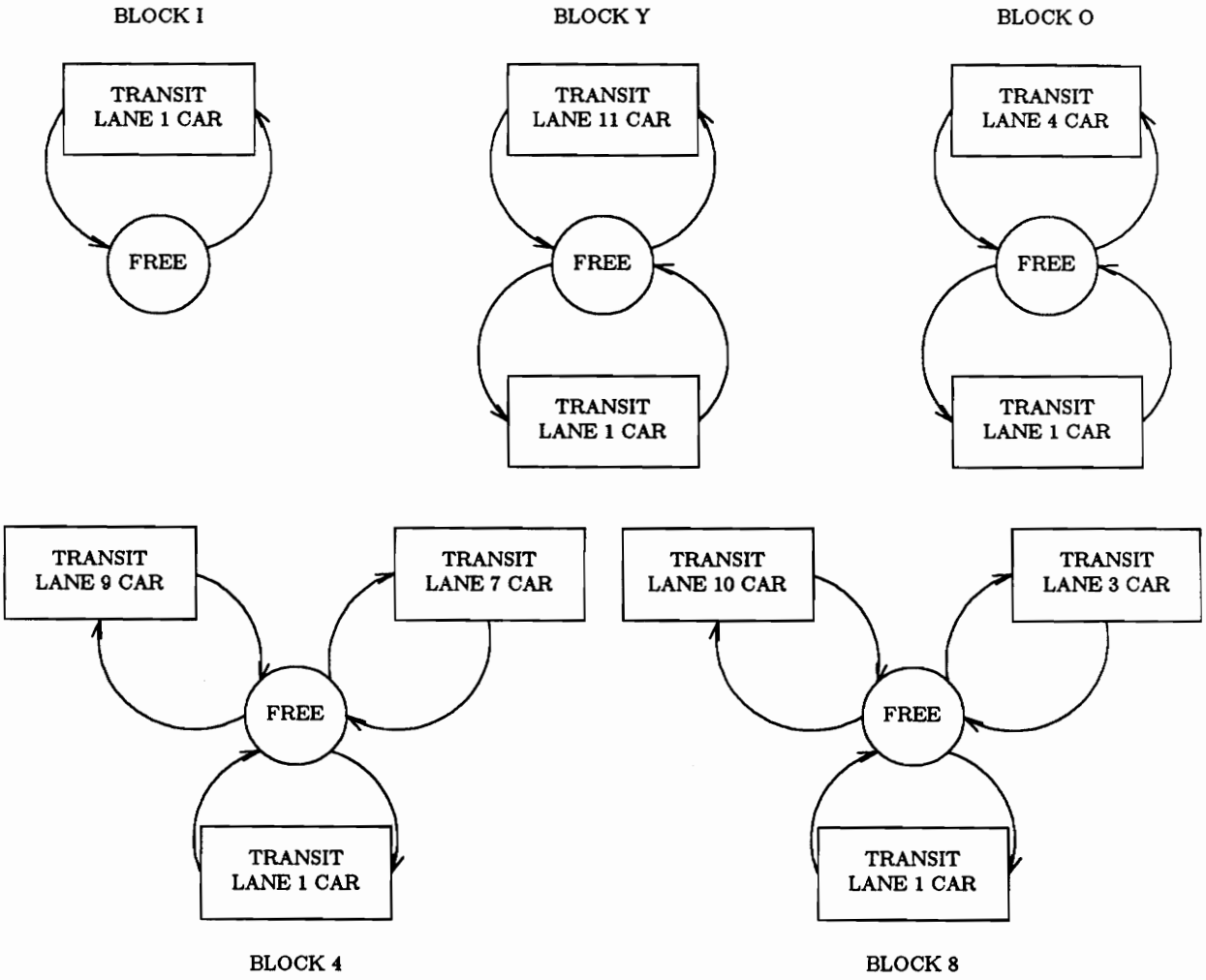


Figure 3.13 Sample of Block Activity Cycle Diagrams

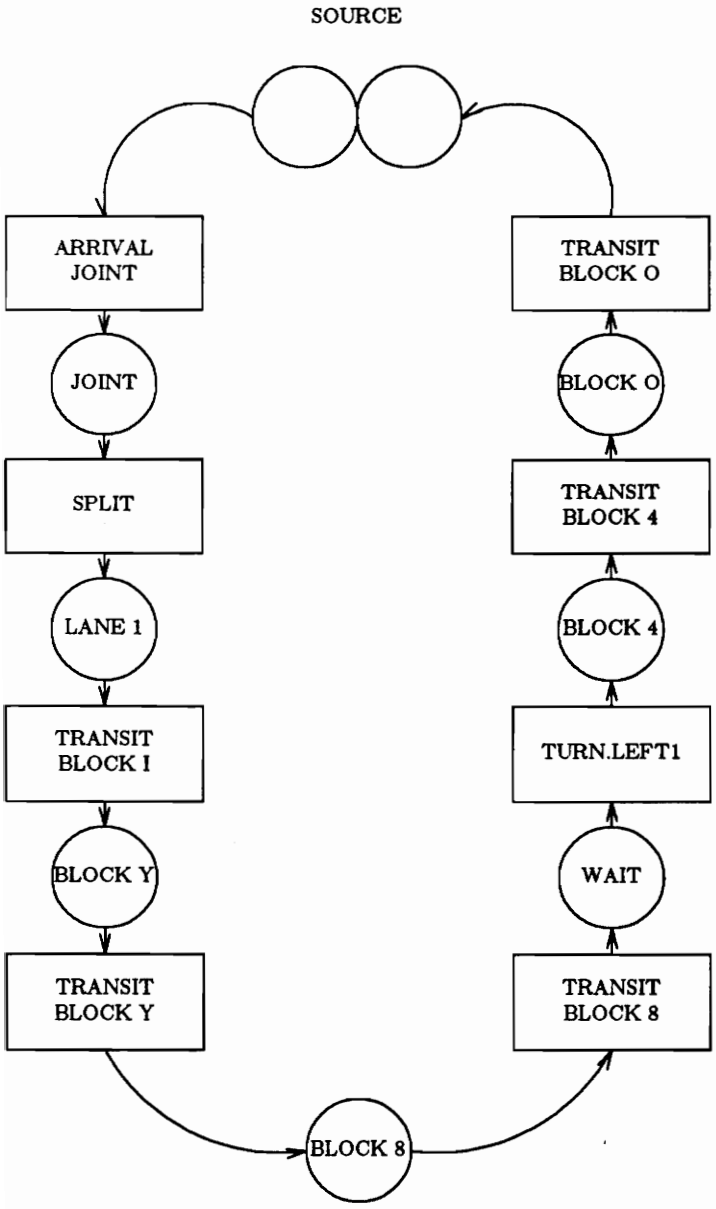


Figure 3.14 Lane 1 Car Activity Cycle Diagram

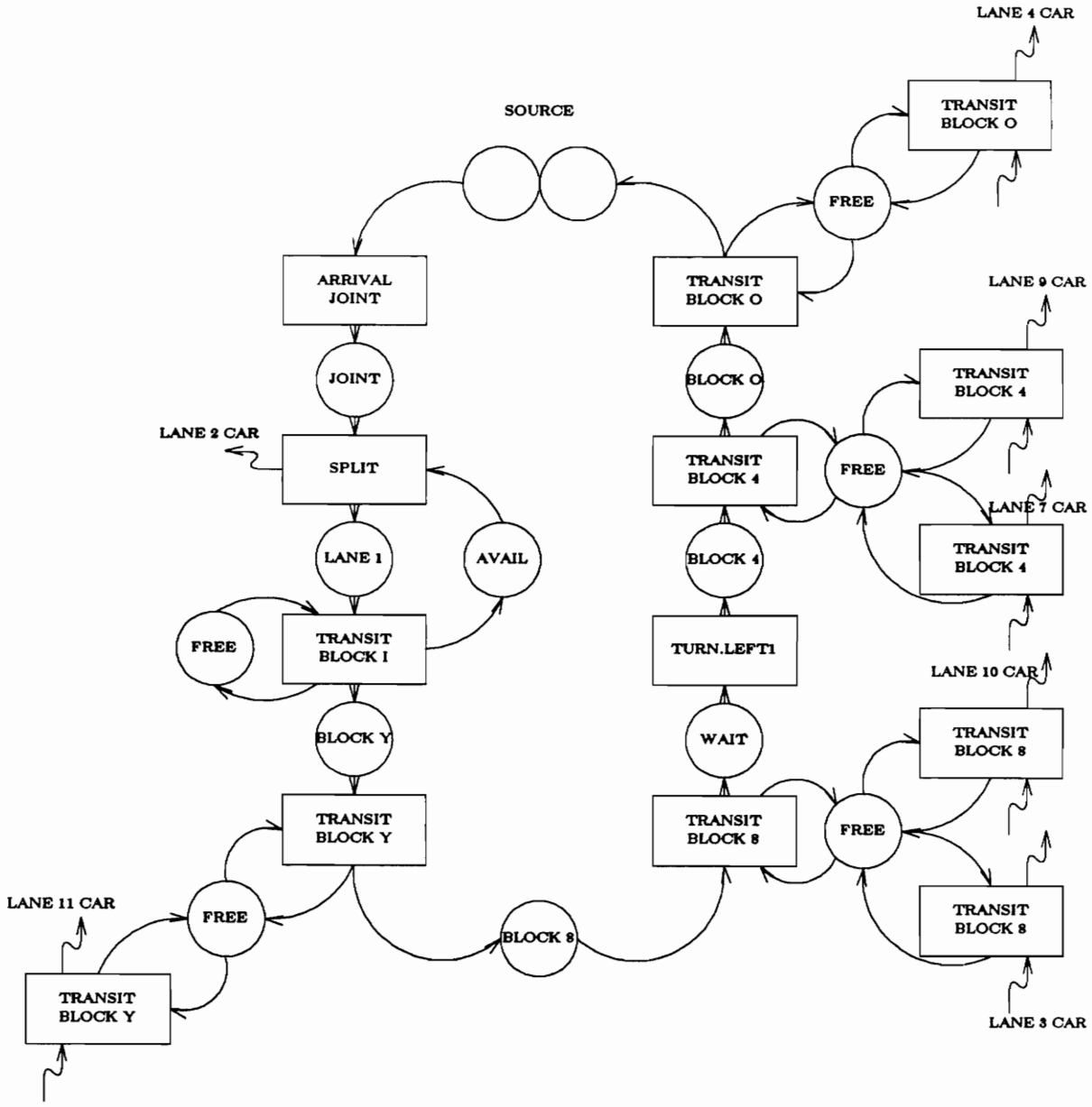


Figure 3.15 Coordinated Activity Cycle Diagram
(Lane 1 Car Path)

JOINT, LANE1, BLOCKY, BLOCK8, WAIT, BLOCK4, and BLOCKO queues are FIFO.
AVAILABLE is a LANE 1 space resource.

3.3.5 Activity Descriptions

To aid the understanding of the following activity descriptions, certain implementation dependent assumptions are listed which underlie these descriptions.

(1) Each of the permanent entities has an associated record in which t-cell values and other necessary information may be stored. For example, the light entity record may allow, in addition to its t-cell value, an indicator of the next light activity. Similarly, the intersection block record may contain fields for the t-cell and a pointer to the car entity (if any) that is transiting the block.

(2) The temporary entities (cars) may also have associated records to store lane identifier, turn, or similar information.

(3) Assume absolute system time is used in t-cells.

(4) The "dueness" of activities (such as activities L1 - L4 or any ETRANS activity discussed below) is determined by a check on the associated entity's t-cell and record information. For example, if the current time equals the t-cell value of the light AND the next light activity is an L1, then activity L1 is due. If the current simulation time equals the t-cell value of the Block I entity, then activity ETRANSI is due.

3.3.5.1 Activity Descriptions associated with the Light

Activity L1 (the North-south green activity)

```
IF L1 is due then begin
      north-south color = green;
      east color = red;
      west color = red;
      intersection previously cleared for west-east traffic = false;
      increment t-cell of the light by 20 seconds;
      identify L2 as light activity which is next due;
      end
ELSE
```

```
        return control to executive;

Activity L2 (the North-south red activity)
    IF L2 is due then begin
        north-south color = red;
        intersection previously cleared for north-south traffic = false;
        increment t-cell of the light by 1 second;
        identify L3 as light activity which is next due;
        end
    ELSE
        return control to executive;

Activity L3 (the West green activity)
    IF L3 is due then begin
        west color = green;
        increment t-cell of the light by 13 seconds;
        identify L4 as light activity which is next due
        end
    ELSE
        return control to executive;

Activity L4 (the East green activity)
    IF L4 is due then begin
        east color = green;
        increment t-cell of the light by 16 seconds;
        identify L1 as light activity which is next due
        end
    ELSE
        return control to executive;
```

3.3.5.2 Activity Description associated with the Arrival Machine

```
Activity ARRJ (Arrival to Joint Lane)
    IF ARRJ is due then begin
        create car (record) for attribute assignment;
        arrival time = current clock time
        select car attribute, Lane 1 or Lane 2;
        IF Lane 2 is selected then
            select additional car attribute, right turn or straight;
        file car in joint lane queue;
        generate time of next ARRJ;
        set t-cell of ARRJ arrival machine to this time;
    ELSE
        return control to executive;
```

3.3.5.3 Special Activity Descriptions

Activity SPLIT (Moving car from Joint Lane to Lane 1 or 2, splitting)

Note that this activity is a dummy activity which is included solely for the purpose of controlling the decision point in a car proceeding into lane 1 or 2. SPLIT consumes zero simulated time. Implementations for this action may vary considerably. The concept of a resource which is covered by O'Keefe and Davies [1987] is utilized. Lane 1 space and Lane 2 space are resources which provide space for 5 (or less) cars in lane1 and in lane 2 respectively. This resource must be available for the SPLIT activity to occur.

```
IF (Joint Lane is not empty and
(((head car in Joint is Lane 1) and (a Lane 1 space is available)) or
((head car in Joint is Lane 2) and (a Lane 2 space is available)))
then begin
      release car from Joint Lane;
      CASE car lane identifier
      Lane 1: begin
            decrement Lane 1 space counter by 1;
            file car in Lane 1 queue
            end;
      Lane 2: begin
            decrement Lane 2 space counter by 1;
            file car in Lane 2 queue
            end;
      end; (* CASE *)
      end (* IF *)
ELSE
      return control to executive;
```

Activity TURN.LEFT.1 (Turn Left in Intersection, Lane 1 Cars)

This activity is also a dummy activity to aid in the decision to turn left. The activity may not be performed if clearance to turn left (explained in the CM application to the TI) is not available.

```
IF Wait1 queue is not empty and
clearance to turn is available then begin
      remove car from Wait1 queue;
      add car to Block 4 queue
```

```
        end (* IF *)
    ELSE
        return control to executive;
```

3.3.5.4 Activity Descriptions associated with Blocks (in Lane 1 Car Path)

Activity BTRANSI (Begin Transit Block I or Enter Intersection at Lane 1)

```
    IF ((north-south color = green) and (block I is free)
    and (Lane 1 queue is not empty) and ((intersection is clear
    for north-south traffic) or (intersection previously cleared)))
    then begin
        IF (intersection not previously cleared) then
            north-south previously cleared = true;
        release car from Lane 1 queue;
        increment Lane 1 space by one;
        block I = busy;
        identify that Block I is being used by this lane 1 car
        increment t-cell of block I by 1.495 seconds
        end
    ELSE
        return control to executive;
```

Activity ETRANSI (End of Transit Block I)

```
    IF ETRANSI is due now for Block I then
        add car to Block Y queue
    ELSE
        return control to executive;
```

Activity BTRANSY (Begin Transit Block Y)

```
    IF Block Y queue is not empty and Block Y is free then begin
        remove car from Block Y queue;
        Block Y = busy;
        IF car is lane 1 car then begin
            identify that Block Y is being used by this lane 1 car;
            Block I = free;
            increment t-cell of Block Y by 1.540 seconds
            end
        ELSE begin      (* Car is lane 11 car *)
            Block W = free;
            identify that Block Y is being used by this lane 11 car
            increment t-cell of Block Y by 0.529 seconds
            end;
        end (* IF *)
    ELSE
        return control to executive;
```

Activity ETRANSY (End Transit Block Y)

```
    IF ETRANSY is due now for Block Y then begin
        IF car in block Y is lane 1 car then
            add car to Block 8 queue
        ELSE  (*car is lane 11 car *)
            add car to Block Z queue
        end  (* IF *)
    ELSE
        return control to executive;
```

Activity BTRANS8 (Begin Transit Block 8)

```
    IF Block 8 queue is not empty and Block 8 is free then begin
        remove car from Block 8 queue;
        Block 8 = busy;
        IF car is lane 1 car then begin
            identify that Block 8 is being used by this lane 1 car;
            increment t-cell of Block 8 by .634 seconds
            end
        ELSE IF car is lane 3 car then
            ... ... ...
        ELSE      (* car is lane 10 car *)
            ... ... ...;
        end (* IF *)
    ELSE
        return control to executive;
```

Activity ETRANS8 (End Transit Block 8)

```
    IF ETRANS8 is due now for Block 8 then begin
        IF car in Block 8 is lane 1 car then
            add car to Wait1 (to turn left) queue
        ELSE if car is lane 3 car then
            ... ... ...
        ELSE      (* car is lane 10 car *)
            ... ... ...
        end (* IF *)
    ELSE
        return control to executive;
```

Activity BTRANS4 (Begin Transit Block 4)

```
    IF Block 4 is not empty and Block 4 is free then begin
        remove car from Block 4 queue;
        Block 4 = busy;
        IF car is lane 1 car then begin
            identify that Block 4 is being used by this lane 1 car;
            Block Y = free;
            increment t-cell of Block 4 by 1.132 seconds
            end
        ELSE IF car is lane 7 car then
            ... ... ...
```

```
        ELSE        (* car is lane 9 car *)
              ... ... ...;
        end (* IF *)
    ELSE
        return control to executive;
```

Activity ETRANS4 (End Transit Block 4)

```
    IF ETRANS4 is due now for Block 4 then begin
        IF car in Block 4 is lane 1 car then
              add car to Block O queue
        ELSE IF car is lane 7 car then
              ... ... ...
        ELSE        (* car is lane 9 car *)
              ... ... ...
        end (* IF *)
    ELSE
        return control to executive;
```

Activity BTRANSO (Begin Transit Block O)

```
    IF Block O queue is not empty and Block O is free then begin
        remove car from block O queue;
        Block O = busy;
        IF car is lane 1 car then begin
              identify that Block O is being used by this lane 1 car;
              Block 8 = free;
              Block 4 = free;
              increment t-cell of Block O by 1.132 seconds
              end
        ELSE        (* car is lane 4 car *)
              ... ... ...;
        end (* IF *)
    ELSE
        return control to executive;
```

Activity ETRANSO (End Transit Block O)

```
    IF ETRANSO is due now for Block O then begin
        IF car in Block O is lane 1 then begin
              Block O = free;
              waiting time for car = current time - arrival time;
              increment counter of departures from lane 1 by 1;
              total waiting time in lane 1 = waiting time for car
                    + previous total waiting time in lane 1;
              end
        ELSE        (* car is lane 4 car *)
              ... ... ...
        end (* IF *)
    ELSE
        return control to executive;
```

3.3.6 Priority of Activities

Activities are prioritized during the activity scan to ensure that model behavior is accurate. The color of the traffic light is the most critical state condition in the model in that it directly influences traffic flow at the intersection. Therefore, the light activities receive top priority among all activities. To ensure that a car may proceed smoothly along its path, end transit activities (e.g. ETRANS8, etc.) are next checked to free as many blocks as possible for cars waiting, delayed in their transit. The processing of arrivals (e.g. ARRJ, etc.) is now performed since the system is free of departing cars and transiting cars have been moved on to their next block. The SPLIT activity follows the Arrival Activities to enable a car that has just arrived to the Joint Lane to be moved on into Lane 1 or 2 if possible. Begin service activities (e.g. BTRANS4, etc.) are now checked to move cars into the intersection or along their way. Finally, TURN.LEFT activities are checked to attempt the further progression of cars in the intersection.

3.4 The TPA CF Application

The discussion of the previous section covering the AS CF application is wholly relevant to the TPA CF application to the TI. We must now simply identify the activities in the AS CF application as B-activities or C-activities. Having done this, the TPA CF executive (described in Section 2.5) may proceed as follows and as suggested by Pidd [1984]. The A Phase, or Time Scan, may be a scan of entity records which include t-cell and a pointer to the next activity for that entity. A once-through scan of these records retrieves the minimum t-cell value of all entities which are due to perform a B-activity next. The simulated clock time is then updated to this minimum value. Along the way, an ordered list of all the B-activities which are due at this time can be constructed. The

executive then executes these activities in order in the B-Phase and proceeds into the C-phase. The type of list used and the specification of the link between the items (B-activity designations) in this ordered list to the B-activity descriptions is strictly an implementation issue that can be handled in a variety of ways. After conducting the C-phase as discussed in Section 2.4, the executive returns to the A-phase and repeats this cycle until simulation termination conditions are reached. Pidd [1984] presents a particularly straightforward implementation of the TPA CF using BASIM, a Three Phase executive written in BASIC.

Several comments to distinguish the TPA CF implementation from that of the AS CF (beyond the clear differences in the executive) can be made. First of all, the testheads can be removed from those activities which are identified as B-activities. It is also important to note that the importance of priorities among activities remains a key issue. The activities on the execution list of B-activities should be ordered by priority. The traditional scan in the C-phase should likewise scan by priority among the C-activities, as before.

Therefore, to complete the description of the AS CF implementation, we need only identify the activities of the AS CF as B-activities or C-activities. Implementing the executive as described above and in Section 2.5 would complete the TPA CF application to the TI. The algorithm for the TPA CF executive has already been given, and examples which illustrate the executive, like BASIM (above), are available in the literature. Therefore, the implementation of the executive will not be covered.

3.4.1 Activity Designations in the TPA CF Application

Activities are specified (below) as **B (Bound)** or **C (Conditional)** activities. In addition, they are listed in a prioritized order. The activity name identifications which are used are the same as those used in the AS CF application.

3.4.2 Listing of B-Activities

Identifier	**Description**
L1	Light, NS green
L2	Light, NS red
L3	Light, West green
L4	Light, East green
ETRANSA	End Transit, Block A
.....	.....
ETRANSZ	End Transit, Block Z
ETRANS1	End Transit, Block 1
.....	.....
ETRANS9	End Transit, Block 9
ARRJ	Arrival, Joint Lane
ARR3	Arrival, Lane 3
.....	.....
ARR11	Arrival, Lane 11

3.4.3 Listing of C-Activities

Identifier	Description
SPLIT	Split car into Lane 1 or 2
BTRANSA	Begin Transit, Block A
.....	.....
BTRANSZ	Begin Transit, Block Z
BTRANS1	Begin Transit, Block 1
.....	.....
BTRANS9	Begin Transit, Block 9
TURN.LEFT.1	Turn left in Lane 1
TURN.LEFT.3	Turn left in Lane 3
TURN.LEFT.6	Turn left in Lane 6
TURN.LEFT.9	Turn left in Lane 9

3.5 The PI CF Application

The PI CF application is demonstrated in this section by a detailed examination of selected portions of a SIMULA [Birtwistle et al. 1979; Franta 1977] model of the TI. SIMULA, like SIMSCRIPT, is an SPL and contains many features which make the programming task of a model (built under the PI CF) a much simpler undertaking. Besides making available the standard statistical packages which include random variate generation from common probability distributions, etc., SIMULA also offers language primitives which enable process development and their coordinated interaction as coroutines. This section discusses these primitives and the essential processes underlying the SIMULA model of the TI. In addition, the SIMULA executive and statistical output routines are

described in order to complement the discussion and add to its completeness.

3.5.1 Key SIMULA Primitives

From the discussion in Section 2.6, the whole of the PI CF is dominated by the concept of the process: its meaning, its construction, and its interactions with other processes. SIMULA offers several language primitives or constructs that directly appeal to these aspects with creative processor control. This discussion is not intended to provide in-depth coverage of SIMULA and will be limited to a description of SIMULA object behavior, in particular, objects of the class PROCESS (For further details, see Section 3.10 or [Franta 1977]). For the purposes of this discussion, objects of the class PROCESS may be considered to be processes that are associated with the objects contained in the model. We shall refer to these processes as object processes.

When the **new** construct is used, an object process data record is created and processor control passes immediately to the action statements or code of that newly created object process [Franta 1977]. Therefore, "new" is somewhat like a typical procedural call. An object process, once generated with "new" construct, may find itself in one of several states (active, passive, suspended, terminated) and may exist as a coroutine which can be executed in a piecemeal fashion. These state categories are defined [Franta 1977] as follows:

- active — executing; only one object process may be active at any one time.
- suspended — owning a notice which is in the sequencing set and scheduled for activation or reactivation; active but delayed in performing its actions.
- passive — not active, suspended, or terminated; action statements are not exhausted or completed but there is no scheduled activation or reactivation time;

delay duration is an unknown quantity; idle; may be activated or reactivated by another object process.

- terminated — action statements have been exhausted; will exist as long as it is referenced; cannot be activated.

Object processes may therefore be *passivated* or *activated.* The **passivate** statement changes the state of an active object process to a passive state and destroys its notice in the sequencing set. A passivated object process experiences a period of conditional delay, awaiting the satisfaction of some "wait-until" condition, if any. The **activate** statement generates a new notice and places that notice in the sequencing set. **Reactivate** cancels an active object process and then activates it. The activation or reactivation of an object process may be dictated when the period of delay is known and unconditional. A passivated object process, when later activated or reactivated, begins the execution of its actions statements following that point at which it was passivated. This "piecemeal" execution continues in this fashion until all action statements are completed.

Therefore, the language primitives, "passivate", "activate", and "reactivate", form the basis for process interaction and behavior within SIMULA. The next section, which covers the primary processes that cooperate within the SIMULA TI model, provides examples of how these primitives are effectively used within process descriptions. Please note that there are lower level primitives (**resume** and **detach**) that are provided by SIMULA. Coverage of these is beyond the scope of this discussion. Franta [1977] provides excellent coverage of process communication using the "resume" and "detach" primitives.

3.5.2 Processes of the SIMULA TI Model

The processes of interest in the SIMULA TI model include a "lightctrl" process which represents the light controller that manages the light timing sequences of the intersection light, a driver process for each vehicle, and the vehicle (or car) processes.

The light controller process is shown in Figure 3.16. The entire process is clearly described within this code. Colors of the north and south directions are initially set to green whereas the east and west directions are red. The process reactivates itself with a 20 second delay being specified. At this time, the north and south directions become red. The **prior** key word ensures that the notice for this process is placed on the sequencing set before all other notices which are to be active at that time. This gives the highest priority to the light controller process. Following another 1 second delay, the west direction becomes green. This state of color is maintained until another reactivation changes the east direction to green, 13 seconds later. A final reactivation after 16 seconds repeats the process cycle since the process is built around the "while true do" looping construct at line 4.

The driver process for each car controls the car's entrance into the transit area of the intersection in the same manner as any real driver does. The driver process for cars that travel into the intersection from the north or south lanes is shown in Figures 3.17 and 3.18. Notice that the driver process checks the appropriate conditions that will allow his car to enter the intersection. These conditions (which depend on light color, block availability, and intersection clearance) are fully described in the CM definition in Section 3.1. If any one condition is not satisfied, the driver will reactivate himself on the sequencing set in the appropriate location. For example, if the light has not permitted entry, the driver places himself after the next light controller activation (line 12). If another

```
*************************************************************************
* DESCRIPTION:  Class definition of the LIGHTCTRL (Light Controller)
*     OBJECT's process.
*
* ATTRIBUTES: lite, the referenced LIGHT OBJECT being controlled.
*
* INPUT(S) :  lite, the LIGHT OBJECT
*
* OUTPUT(S):  "red" or "green" status is output to the directions of
*     the LIGHT OBJECT to simulate the light timing sequences.
* CALLS    :  No procedural calls, but remote access of the LIGHT
*     OBJECT is performed.
* CALLED BY:  Referenced by GENOBJECTS upon creation.
*************************************************************************;
1  process class LIGHTCTRL(lite);
2  ref(light)lite;
3  begin
4     while true do
5        begin
6           lite.north.setgreen;              !Set north and south green  ;
7           lite.south.setgreen;
8           lite.east.setred;                 !Set east and west red      ;
9           lite.west.setred;
10          reactivate this lightctrl delay 20 prior;
11          lite.north.setred;                ! Set north and south to red;
12          lite.south.setred;                !  for 1 sec clearance.     ;
13          reactivate this lightctrl delay 1 prior;
14          lite.west.setgreen;               ! Set West to green         ;
15          reactivate this lightctrl delay 13 prior;
16          lite.east.setgreen;               ! Now East set to green     ;
17          reactivate this lightctrl delay 16 prior;
18       end;                                 ! Now restart the cycle      ;
19    end CLASS LIGHTCTRL;
```

Figure 3.16 The LIGHTCTRL Object Process

```
*************************************************************************
* DESCRIPTION: Class Definition of a NS DRIVER OBJECT (North, South)
*
* ATTRIBUTES: The traffic light (mylight), the driver's car(mycar),
*   the first block to enter (myblock), and the intersection (road).
*
* INPUT(S) : The above refereced attributes are input on driver creation.
*
* OUTPUT(S): The driver activates his car at the appropriate times
*   after successful checks for entrance to the intersection OR after
*   successful check for a left turn (if applicable).  Otherwise, the
*   driver picks an appropriate place to reactivate himself to check
*   for the right conditions to put his car in motion.  The driver
*   effectively provides a "wait-until" capability.
*
* CALLS    : The appropriate intersection (road) clearance routine.
*
* CALLED BY: The driver is activated by his car to enter the inter-
*   section or to turn left once in the intersection.
*************************************************************************;
driver class NSDRIVER(mylight, myblock, road, mycar);
1  ref(light)mylight; ref(block)myblock; ref(intersection)road;
2  ref(car)mycar;
3  begin
4     inspect mycar do begin
5        if not entered then begin           ! Check for entry to inters.;
6  start: if mylight.north.red then begin    ! First check light          ;
7           road.clearedns := false;         !   Set road not clear for ns;
8           if (lane= 2 and right) or lane = 8 then begin
9              goto block;                   ! Right turner may continue ;
10             end
11          else begin
12             reactivate this nsdriver after controller;
13             goto start;                   ! Restart the check at light;
14             end;
15          end;
16 block: if myblock.busy then begin         ! Next check the first block;
17          place(this nsdriver);
18          goto start;
19          end;
20       if mylight.north.red and lane eq 2 and right then begin
21          if not road.r2clear then begin  ! This check for a turner    ;
22             place(this nsdriver);        !  from lane 2               ;
23             goto start;
24             end;
25          end
26       else if mylight.north.red and lane = 8 then begin
27          if not road.r8clear then begin  ! This check for a turner    ;
28             place(this nsdriver);        !  from lane 8               ;
29             goto start;
30             end;
31          end
32       else if ((not road.clearedns) and (not road.nsclear))
33          then begin                      !Now check intersection clear;
34          place(this nsdriver);
35          goto start;
36          end;
37       activate mycar after current;
38       passivate;
```

Figure 3.17 NSDRIVER Process

```
39       if lane = 1 then begin              !These checks for left turns ;
40          while not road.left1ok do begin
41             place(this nsdriver); end;
42          end
43       else begin
44          while not road.left6ok do begin
45             place(this nsdriver); end;
46          end;
47       activate mycar after current;
48       passivate;
49       end
50    else begin                             !Left turn checks, for cars ;
51       if lane = 1 then begin              ! that immediately entered  ;
52          while not road.left1ok do begin  ! without initial need of   ;
53             place(this nsdriver); end;    ! driver.                   ;
54          end
55       else begin
56          while not road.left6ok do begin
57             place(this nsdriver); end;
58          end;
59       activate mycar after current;
60       passivate;
61       end;
62    end;
63 end CLASS NSDRIVER;
```

Figure 3.18 NSDRIVER Process (continued)

condition has denied entry, then the user-defined **place** procedure (lines 17, 22, 28, and 34) inserts the driver process in the sequencing set after the next non-driver process (the next possible process that might change state conditions). Following such placement, **go to** is used to restart the check of conditions at the **start** label, line 6. Once the entrance conditions are satisfied, the driver will immediately activate his car process and will passivate himself. In most cases, his job has been completed. Cars travelling in lanes 1 and 6, however, have their drivers reactivated to check for left turn clearance of oncoming traffic. See lines 39-63 which cover these instances.

A car process describes the complete movement of the car including arrival to the intersection, transit of the intersection, and subsequent departure. Figure 3.19 describes the basic actions of **all** car processes. There are arriving actions such as recording the arrival time (lines 14-16), then specific lane functions which are accomodated by the **inner** key word (line 17), and finally departure actions (lines 19-39) to record data for later statistical and performance measure calculations. Additional code, determined by the lane association of the car, is essentially inserted at the "inner" construct and represents the specific lane functions of the car. Figure 3.20 represents an example of this "additional" code which, in this case, is the specific lane function for a car in lane 8. Here in lines 3-19, a user-defined procedure **transitfm8** gives the specific details of the actions that are performed by a car transiting the intersection from lane 8. The **setbusy** and **setfree** procedures update the busy or free status of the blocks that are crossed during the car's transit. Therefore, it is clear that blocks "a" and "k" provide the path for a lane 8 car. Notice that if block "k" is busy (line 8), the object process is placed on a queue and passivated (lines 9 and 10) until the block becomes free and the object process is at the head of the queue ("blockqk") of processes waiting for that block. Also, once a car has

```
*************************************************************************
* DESCRIPTION: Class definition for a CAR OBJECT
* ATTRIBUTES: Arrive and depart procedures.
*             arrtime - arrival time
*             lane    - resident lane of car.
*             id      - car id number for trace purposes.
*             right   - boolean indicating if car is right turner.
*             entered - boolean indicating if car is "in" intersection.
* INPUT(S) :  None
* OUTPUT(S):  Statistics information on departure to waiting time and
*    departure variables.
* CALLS    :  procedure update upon departure to enter statistics.
* CALLED BY:  Referenced by GENOBJECTS upon creation.
*************************************************************************;
process class CAR;
1  begin
2     real arrtime;                          ! Car arrival time          ;
3     integer lane;                          ! Resident lane of car      ;
4     integer id;
5     boolean right;                         ! Right turn boolean        ;
6     boolean entered;                       ! In or out of intersection ;
7     procedure update(waittime, departures);! Update lane waiting time  ;
8        name waittime, departures;          !  and departures in lane   ;
9        real waittime; integer departures;
10       begin
11          waittime := waittime + time - arrtime;
12          departures := departures + 1;
13       end PROCEDURE UPDATE;
14    id := ctr + 1;                         ! Arriving actions          ;
15    ctr := ctr + 1;                        ! Set id and next id counter.;
16    arrtime := time;                       ! Set arrival time of car.  ;
17    inner;                                 ! Do specific lane functions ;
18                                           ! Now do terminating actions ;
19    if lotp > 1 then                       !    When in transient pd.  ;
20       lotp := lotp - 1                    ! OR                        ;
21    else begin                             !    When entering s. s.    ;
22       if lotp = 1 then
23          lotp := 0                        ! OR                        ;
24       else begin                          !    When in s. s.          ;
25          ndiss := ndiss + 1;             !      count departures and ;
26          if lane = 1 then                !      update waiting time  ;
27             update(twt1, deps1)          !      and # departures     ;
28          else if lane = 2 then begin     !      based on lane that car;
29             if right then                !      was in.              ;
30                update(twt2r, deps2r)
31             else
32                update(twt2, deps2)
33          ...   ...   ...   ...   ...
34          else if lane = 11 then begin
35             if right then
36                update(twt11r, deps11r)
37             else
38                update(twt11, deps11)
39             end
40          else;
41          if ndiss = loss then begin     ! Now departures indicate   ;
42             activate main;              !  replication is over so   ;
43             passivate;
44             end                         !  activate the main program.;
45          end
46       end
47 end CLASS CAR;
```

Figure 3.19 Generic Car Process

```
*************************************************************************
* DESCRIPTION: Class description for a CAR8 OBJECT, ie a car in
*              See description for CAR1_2 OBJECT since very similar.
*              Inline code comments from CAR1_2 also pertain.
* ATTRIBUTES:  Procedure transitfm8 gives process description for cars
*   transiting from lane 8.
*              mydriver - the driver process for the car
* INPUT(S)   : Transit procedure is given the intersection object
*
* OUTPUT(S) :  No direct output.
*
* CALLS      : Attribute setting procedures of each block transited.
*              (setbusy, setfree)
* CALLED BY :  Referenced by main simulation routine upon creation.
*************************************************************************;
car class CAR8;
1  begin
2     ref(nsdriver) mydriver;
3     procedure transitfm8(road);
4     ref(intersection)road;
5     begin
6     square_a.setbusy(this car8);              !Enter and transit A      ;
7     reactivate this car8 delay(square_a.findtransit(this car8));
8     if square_k.busy then begin               !Check K, queue up if busy;
9        into(blokqk);
10       passivate;
11       out;
12       end;
13    square_k.setbusy(this car8);              !Transit K                ;
14    square_a.setfree;                         !Release A                ;
15    reactivate this car8 delay(square_k.findtransit(this car8));
16    square_k.setfree;                         !Release K                ;
17    if not blokqk.empty then
18       activate blokqk.first after current;   !Enable cars waiting for K;
19    end TRANSITFM8;
    comment;
20    activate new car8 delay (weibl(56.0592, 0.63923, seed8));
                                                !Generate next arrival    ;
21    mydriver :- new nsdriver(tfclight, square_a, pforkandtcreek,
22          this car8);                         !Create driver            ;
23    lane := 8;                                !Set attributes           ;
24    right := true;
25    if not lane8.empty then begin             !Enter lane8 queue when   ;
26       into(lane8);                           ! cars are already in lane;
27       passivate;                             !Wait in line for turn    ;
28       activate mydriver after current;       !At head of line, turn on ;
29       passivate;                             ! driver.                 ;
30       out;
31       end
32    else if (((tfclight.south.red) or (square_a.busy) or
        ((not pforkandtcreek.nsclear) and
        (not pforkandtcreek.clearedns))) and
        ((not right) or (tfclight.south.green) or (square_a.busy) or
        (not pforkandtcreek.r8clear))) then begin
33       into(lane8);                           !Can't immediately enter  ;
34       activate mydriver after current;       ! so first in queue,      ;
35       passivate;                             ! and turn on driver.     ;
36       out;
37       end;
38    if not lane8.empty then                   !Ready to enter, so turn  ;
39       activate lane8.first after current;    ! on any car waiting in   ;
40    entered := true;                          ! lane8 queue.            ;
41    transitfm8(pforkandtcreek);               !Enter and transit        ;
42 end CLASS CAR8;
```

Figure 3.20 The CAR8 Process

entered a block (such as the case in lines 13-15), the block is set to "busy" and an unconditional delay, corresponding to the transit time across the block, is set with a reactivation statement. The bootstrapping of future arrivals to lane 8 (line 20), the creation of the car's associated driver process (line 21), and the setting of the lane identification and turn indication attributes (lines 23,24) is included in Figure 3.20. This figure also shows that if conditions are right, that is if the conditional traps at lines 25 and 32 are passed, the car may immediately proceed into the intersection. Otherwise, the car is queued up in the appropriate lane queue, its driver is activated, and the process is passivated (until later activated by its driver).

An important aspect of the above discussion of the processes in the SIMULA model is that the modeler is required to maintain control of process activation and passivation. This adds to the complexity and difficulty of the modeling task.

3.5.3 The SIMULA Executive

The executive, Figure 3.21, is itself a process which first performs initializations within the **setup** routine (line 13), creates the necessary object processes via the **genobjects** routine (line 14), schedules the initial arrivals to each lane in lines 15-25, and then passivates itself. The last car to depart the intersection (satisfying simulation termination conditions, see Figure 3.19, lines 41-43) activates the executive which then performs the statistical output actions. The entire executive process is surrounded by a looping construct (line 4) that indexes on the number of replications, thereby accomplishing the method of replications to achieve the desired results of the simulation study objectives.

```
*************************************************************************
* DESCRIPTION: Traffic intersection simulation using process view.
*      Demonstrates the use of the process interaction world view with
*      an example simulation containing sufficient complexity to show
*      the characteristics which are embodied in the view.  The model
*      is a simulation of the intersection of Prices Fork Rd. and Toms
*      Creek Road near the campus of Virginia Tech, Blacksburg, Va.
*
* ATTRIBUTES: Not applicable
*
* INPUT(S) : Random variates from external FORTRAN routines,
*     WEIBL, GAMA, EXPON, and proper random numbers from RANDM.
* OUTPUT(S): Performance measures for average waiting times for cars in
*     in each of the eleven lanes and also in lanes 2, 5, and 11 when
*     right turns are being made.  The performance measures are output
*     after each replication.
*
* CALLS     :   Within the simulation block- Procedures SETUP,
*     GENOBJECTS, and STATISTICS.
* CALLED BY: User upon execution of the model.
*************************************************************************;
    ...    ...    ...    ...    ...
1   integer numruns;                          ! Number of simulation runs  ;
2   integer numrng;                           ! Number of random generators;
    ...    ...    ...    ...    ...
3   comment
    ******** BEGIN SIMULATION MODEL AND LOOPING FOR REPLICATIONS **********;
4   for i := 1 step 1 until numruns do
5    simulation begin
     ...    ...    ...    ...    ...
6    real twt1, twt2, twt2r, twt3, twt4, twt5; ! Total waiting time per lane;
     ...    ...    ...    ...    ...
7    integer deps1, deps2, deps2r,deps3, deps4;! Departures in s.s. per lane;
     ...    ...    ...    ...    ...
8    external FORTRAN real procedure randm;    ! Generator of rn's, 0 to 1  ;
9    external FORTRAN real procedure weibl;    ! Generator for lanes 4,7,8  ;
10   external FORTRAN real procedure gama;     ! Generator for lane 3       ;
11   external FORTRAN real procedure expon;    ! Generator for lane 6       ;
     ...    ...    ...    ...    ...
12    comment
      MAIN SIMULATION PROGRAM;
      ...    ...    ...    ...    ...
13    setup(i);                                  ! Setup for replication      ;
14    genobjects;                                ! Create all model objects   ;
15    activate controller after current;         ! Generate first arrivals    ;
16    activate new car12 delay (linear(m1_2a, m1_2b, seed1_2));
17    activate new car3 delay (gama(51.248, 1.260, seed3));
18    activate new car4 delay (weibl(10.6646, 0.82821, seed4));
19    activate new car5 delay (linear(m5a, m5b, seed5));
20    activate new car6 delay (expon(54.6774, seed6));
21    activate new car7 delay (weibl(34.7083, 0.86424, seed7));
22    activate new car8 delay (weibl(56.0592, 0.63923, seed8));
23    activate new car9 delay (linear(m9a, m9b, seed9));
24    activate new car10 delay (linear(m10a, m10b, seed10));
25    activate new car11 delay (linear(m11a, m11b, seed11));
26    passivate;
27    statistics;                               !Output statistics           ;
28    outtext("***************END RUN***************"); outimage;
29    end SIMULATION RUN;
    ...    ...    ...    ...    ...
30   end MODEL;
```

Figure 3.21 The SIMULA Executive or Main Routine

3.5.4 The Statistical Output Routine

The Statistical output routine, Figure 3.22, initializes the performance measure variables (average or mean waiting times of cars, by lane) and then calculates their value by dividing the total waiting time of all cars in a particular lane by the number of departures of cars in that lane. Output is sent to three separate files. An example of the output for a model execution of three replications is shown in Figure 3.23.

3.6 The TF CF Application

In this section, the discussion centers on the block structure of a GPSS/H [Henriksen and Crain 1983] model of the TI and the organization of its block structure to formulate the model processes. After a short introduction to the model, the central model segments or submodels that make up the model processes are reviewed. This part of the discussion includes a description of the light submodel and an example lane submodel, in this case, from lane 8. Using lane 8 as the example will enable the reader to compare this portion of the GPSS/H model with the SIMULA model's corresponding code (Figure 3.20). Finally, the overall model executive and its statistical output are covered.

3.6.1 Introduction to the GPSS/H Model

GPSS/H is a widely used SPL which is based on the TF CF. The GPSS/H model used in this section was developed by Osman Balci, and is a useful example for informatively demonstrating the TF CF with its block-oriented nature and its use of transactions which "flow" through the model segments. The GPSS/H model conforms to the description of the CM definition given in Section 3.1 of the TI and supports the listed objectives.

A general description of the model is displayed in Figure 3.24. Model background,

```
*************************************************************************
* DESCRIPTION: STATISTICS procedure wraps up all statistical infor
*     for output at the end of each simulation run.
*
* ATTRIBUTES: None
*
* INPUT(S) : None
* OUTPUT(S): Performance measures, average waiting times per lane.
*
* CALLS    : None
*
* CALLED BY: Replication within main simulation program.
*************************************************************************;
procedure STATISTICS;
1  begin
2     ref(car)temp;
3     real awt1, awt2, awt2r, awt3, awt4;     !Ave. waiting times for each;
4     real awt5, awt5r, awt6, awt7, awt8;     ! lane.                      ;
5     real awt9, awt10, awt11, awt11r;
6     awt1 := 0; awt2 := 0; awt3 := 0;
7     awt4 := 0; awt5 := 0; awt6 := 0;
8     awt7 := 0; awt8 := 0; awt9 := 0;
9     awt10 := 0; awt11 := 0;
10    awt2r := 0; awt5r := 0; awt11r := 0;
11    if deps1 ne 0 then                      !Calculate perf measures when;
12       awt1 := twt1/deps1;                  ! there have been departures ;
13    if deps2 ne 0 then                      ! from a lane.               ;
14       awt2 := twt2/deps2;
15    if deps2r ne 0 then
16       awt2r := twt2r/deps2r;
      ...   ...   ...   ...   ...
17    if deps11 ne 0 then
18       awt11 := twt11/deps11;
19    if deps11r ne 0 then
20       awt11r := twt11r/deps11r;           !Output stats to files       ;
21    one.outfix(awt1,4,10); one.outfix(awt2,4,10); one.outfix(awt2r,4,10);
22    one.outfix(awt3,4,10); one.outfix(awt4,4,10); one.outfix(awt5,4,10);
23    one.outimage;
24    two.outfix(awt5r,4,10); two.outfix(awt6,4,10); two.outfix(awt7,4,10);
25    two.outfix(awt8,4,10); two.outfix(awt9,4,10); two.outfix(awt10,4,10);
26    two.outimage;
27    three.outfix(awt11,4,10); three.outfix(awt11r,4,10);
28    three.outimage;
      ...   ...   ...   ...   ...
29 end STATISTICS;
```

Figure 3.22 The STATISTICS Routine

```
6
Average waiting time in lane 1
Average waiting time in lane 2 (str)
Average waiting time in lane 2 (rt)
Average waiting time in lane 3
Average waiting time in lane 4
Average waiting time in lane 5(str)
   19.3898   17.2164   10.9081   19.7511   17.3328   18.0267
   19.6043   16.9220   11.0476   19.3447   17.7640   17.9730
   19.7130   17.1718   10.6509   19.5991   17.8314   17.7925


6
Average waiting time in lane 5(rt)
Average waiting time in lane 6
Average waiting time in lane 7
Average waiting time in lane 8
Average waiting time in lane 9
Average waiting time in lane 10
   13.2123   19.4683   16.4761    5.3249   13.1209    9.5287
   13.1906   20.3019   15.4443    4.6897   12.7754    9.8429
   12.9573   19.8090   15.4271    5.4018   12.7706    9.8029


2
Average waiting time in lane 11
Average waiting time in lane 11(rt)
   11.6123    8.3418
   11.5982    8.7260
   11.6678    8.1801
```

Figure 3.23 Output of Three Replications of SIMULA Model

```
*************************************************************************
*
*                    A GPSS/H SIMULATION MODEL OF THE
*                        TRAFFIC INTERSECTION AT
*                    PRICES FORK AND TOMS CREEK ROADS
*
* DESCRIPTION:  A study was initiated as a term project in CS 4150
*               during Winter 1987 quarter. The objective of the study
*               was to compare the current light timing with two other
*               alternative ones to see if the average waiting times
*               of vehicles can be reduced to an acceptable level.
*               The whole class participated in data collection and
*               UNIFIT package program was used to analyze the data.
*               This is a GPSS/H model of the traffic intersection.
*
* HISTORY
*   Created By     : Osman Balci
*   Date Created   : 3 June 1987
*   Revised By     :
*   Date Revised   :
*   Revision Notes:
*
* INPUTS:      none
*
* OUTPUTS:     - GPSS/H standard output for replications 1, 2,
*                &NRUNS-1, and &NRUNS.
*              - FILE FT09F001 A1 file containing the confidence
*                intervals for the 14 performance measures
*
* CALLS:       - CISUB : Confidence Interval construction
*                        SUBroutine (FORTRAN)
*              - GAMA  : Gamma random variate generator (FORTRAN)
*              - WEIBL : Weibull random variate generator (FORTRAN)
*              - EXPON : Exponential random variate generator (FORTRAN)
*
*                [Note : GAMA, WEIBL, and EXPON call RAND random
*                         number generator (FORTRAN)]
*
* ACTIVATION:  gpssh tomscpf size=c
*
*************************************************************************
*
*        Time Unit = Milliseconds
*
         SIMULATE                  Compile, Link, Load, and Run
*
         OPERCOL  30               OPERand start COLumn <= 30
*
*                                   CI  = Confidence Interval
*                                   RVG = Random Variate Generator
*
         EXTERNAL   &CISUB         CI construction SUBroutine in FORTRAN
         EXTERNAL   &GAMA          Gamma RVG in FORTRAN
         EXTERNAL   &WEIBL         Weibull RVG in FORTRAN
         EXTERNAL   &EXPON         Exponential RVG in FORTRAN
*
         INTEGER    &I,&J          Index Variables Used in DO Loops
         INTEGER    &NRUNS         No. of Simulation Runs (Replications)
*
```

Figure 3.24 GPSS/H Model Description, Declarations, and Initiation

objectives, initial declarations, and the GPSS initiation sequence are all included in this figure. Figure 3.25 records the declaration of the performance measure variables. Fourteen arrays, distinguished in name by the lane which they represent, hold the average waiting times of all cars in that lane. The initialization of sets of random seed values, used to recreate the exactly same seed streams and experimental conditions during test runs, is also included. Figure 3.25 concludes with a description of the assignments of these seed sets, stored in a two-dimensional array **MX$SEED**, to appropriate random number generators. Taken as a whole, Figures 3.24 and 3.25 provide a summary of the basis of the GPSS/H model of the TI with insight into some of its implementation details.

The primary interest in this GPSS/H model is to use it to illustrate the distinguishing features of the TF CF. The model is composed of various submodels or groupings of code by function, in particular it includes

- the LIGHT submodel,
- an example LANE submodel (LANE 8),
- the EXPERIMENTAL CONTROL submodel, and
- the CI (Confidence Interval) CONSTRUCTION submodel.

The LIGHT and LANE submodels are model segments which are derived from the GPSS/H block statements. Since GPSS/H is an extension of the PI CF, these two submodels are actually process descriptions with a material-oriented perspective, and are clearly the most illustrative of the TF CF features. The EXPERIMENTAL CONTROL submodel and the CI CONSTRUCTION submodels are both formed from GPSS/H control statements. The EXPERIMENTAL CONTROL submodel serves as the executive of the model. **CALL** control statements to an external FORTRAN routine CISUB delivers the

```
*-------------------------------------------------------------------
* There are 14 performance measures (response variables) as defined
* below. The Ith element of the array contains the Average Waiting
* Time Of Vehicles (AWTOV) in a path of traveling and is obtained from
* the Ith replication of the simulation model.
*-------------------------------------------------------------------
*
         REAL        &AWTOV1L(30)  AWTOV Turning Left from Lane 1
         REAL        &AWTOV2S(30)  AWTOV Traveling Straight from Lane 2
         REAL        &AWTOV2R(30)  AWTOV Turning Right from Lane 2
         REAL        &AWTOV3L(30)  AWTOV Turning Left from Lane 3
         ...        ...    ...    ...    ...    ...    ...
         REAL        &AWTOV10S(30) AWTOV Traveling Straight from Lane 10
         REAL        &AWTOV11S(30) AWTOV Traveling Straight from Lane 11
         REAL        &AWTOV11R(30) AWTOV Turning Right from Lane 11
*
         CHAR*80     &TITLE        Title of a performance measure
*
         LET         &NRUNS=30     Number of Runs (Replications) = 30
*
SEED     MATRIX      MX,&NRUNS,14  Two-dimensional array containing
*                                  seeds for random number streams
*
*-------------------------------------------------------------------
*       Initialization of MX$SEED with random seed values
*-------------------------------------------------------------------
*
         DO          &J=1,14
            DO          &I=1,&NRUNS
               INITIAL     MX$SEED(&I,&J),(13579*RN1)
            ENDDO
         ENDDO
*
*-------------------------------------------------------------------
* Since this is a study of comparing different light timings for the
* traffic intersection, exactly the same experimental conditions must
* be used for all alternative light timings corresponding to a repli-
* cation of the simulation run.  (RNG = Random Number Generator)
* RNj's are the GPSS/H internal RNGs. RAND is the FORTRAN RNG.
*-------------------------------------------------------------------
* RNG      Seed Value                    Used For
* ----   --------------  ------------------------------------------
* RN1        default       Generating seed values for MX$SEED
* RN2     MX$SEED(&I,1)    Generating vehicle arrivals to Lanes 1 & 2
* RN3     MX$SEED(&I,2)    Probabilistic branching to Lane 1 or 2
* RN4     MX$SEED(&I,3)    Probabilistic right turn or straight from 2
* RAND    MX$SEED(&I,4)    Generating vehicle arrivals to Lane 3 (GAMA)
* RAND    MX$SEED(&I,5)    Generating vehicle arrivals to Lane 4 (WEIBL)
* RN5     MX$SEED(&I,6)    Generating vehicle arrivals to Lane 5
* RN6     MX$SEED(&I,7)    Probabilistic right turn or straight from 5
* RAND    MX$SEED(&I,8)    Generating vehicle arrivals to Lane 6 (EXPON)
* RAND    MX$SEED(&I,9)    Generating vehicle arrivals to Lane 7 (WEIBL)
* RAND    MX$SEED(&I,10)   Generating vehicle arrivals to Lane 8 (WEIBL)
* RN7     MX$SEED(&I,11)   Generating vehicle arrivals to Lane 9
* RN8     MX$SEED(&I,12)   Generating vehicle arrivals to Lane 10
* RN9     MX$SEED(&I,13)   Generating vehicle arrivals to Lane 11
* RN10    MX$SEED(&I,14)   Probabilistic right turn or straight from 11
*-------------------------------------------------------------------
```

Figure 3.25 Performance Measure Variables and Seed Initializations

final output of the model, 95 per cent confidence intervals, calculated from the data set stored in the performance measure arrays.

3.6.2 The LIGHT and LANE Submodels

The LIGHT submodel, Figure 3.26, creates (with the **GENERATE** statement) a single transaction to represent the light controller. As this transaction flows through the model segment, block statements are used to easily describe its process. **ADVANCE** statements enable the strict timing control of color (state) changes for the light. These state changes are accomplished with the **LOGIC** switches in which boolean **R** and **S** (red and green) values are associated with each light direction (**LYTESN** — North, South and South, North; **LYTEEW** — East, West; and **LYTEWE** — West, East). A diagram shows the timing sequence of color changes by direction. The process is placed in a cycle with the unconditional **TRANSFER** to the **REPEAT** label.

The LANE submodel, Figure 3.27, perhaps most closely demonstrates the use of blocks and the flow of transactions. Representative of the other LANE submodels (Note that there are eleven others, including LANE1, LANE2, etc.), the LANE8 submodel "generates" a transaction representing a vehicle arriving to Lane 8 at timed intervals determined by the Weibull distribution as shown. The single GENERATE statement accomplishes the bootstrapping of future arrivals. Each transaction, once created, will then "flow" through the block statements of this model segment, effectively simulating the behavior of a Lane 8 vehicle in transit. A boolean variable, **ENTER8R**, is used to specify the conditions necessary for entering the intersection from Lane 8; it is tested with the **TEST** statement to check for proper conditions. The **SEIZE** and **RELEASE** statements are effectively used to the modeler's advantage to move the transaction from the

```
*-------------------------------------------------------------------*
*            L I G H T   T I M I N G   S U B M O D E L              *
*-------------------------------------------------------------------*
*                                                                   *
*   LIGHT TIMING AT PRICES FORK AND TOMS CREEK TRAFFIC INTERSECTION *
*               (Time values are in seconds)                        *
*                                                                   *
*   Direction: North to South and South to North
*
*     Lanes          green          red              red
*   1,2,6,7,8 |--------------------|-|------------------------------|
*                     20            1               29
*
*   Direction: East to West
*
*       Lanes          red          red    red           green
*       3,4,5 |--------------------|-|------------|----------------|
*                     20            1     13             16
*
*   Direction: West to East
*
*      Lanes          red          red           green
*     9,10,11 |--------------------|-|-----------------------------|
*                     20            1              29
*
*   Assumption:
*
*   (1)  Yellow light is included in green.
*
*-------------------------------------------------------------------
*
          GENERATE    ,,,1,1        Generate one transaction representing
*                                   light controller with a priority of 1
*                                   higher than the priorities of vehicles
REPEAT    LOGIC S     LYTENSN       Light for NS & SN directions is green
          LOGIC R     LYTEEW        Light for EW direction is red
          LOGIC R     LYTEWE        Light for WE direction is red
          LOGIC R     CLEARNSN      Intersection clearance has been checked
*                                   for NS & SN traffic when light LYTENSN
*                                   just turns green
          ADVANCE     20000         Lights stay in this status for 20 secs
*
          LOGIC R     LYTENSN       Light for NS & SN directions is red
*
          ADVANCE     1000          One-second intersection clearance
*
          LOGIC S     LYTEWE        Light for WE direction is green
          LOGIC R     CLEARWE       Intersection clearance is not checked
*                                   for West to East traffic when light
*                                   LYTEWE just turns green
          ADVANCE     13000         Lights stay in this status for 13 secs
*
          LOGIC S     LYTEEW        Light for EW direction is green
*
          ADVANCE     16000         Lights stay in this status for 16 secs
*
          TRANSFER    ,REPEAT       Start a new cycle of light timing
*
```

Figure 3.26 LIGHT Submodel

```
*-------------------------------------------------------------------------
*                      L A N E 8     S U B M O D E L
*-------------------------------------------------------------------------
* Using UNIFIT package program, interarrival times of vehicles to
* Lane 8 have been found to fit to a WEIBULL probability distribution
* with the following parameter values:
*
* Location Parameter -   0.                Mean      -    36.8298
* Scale    Parameter -   56.0592           Variance  -  1756.41
* Shape    Parameter -   0.63923
*-------------------------------------------------------------------------
*
*-------------------------------------------------------------------------
* For the vehicle at the front end of Lane 8 to turn right:
*
*   If LYTENSN is green, then [ block A must be empty
*         AND (if LYTENSN has just turned green, then the intersection
*                should first be cleared for the NS & SN traffic)
*   ] else [ blocks A, K, L, H, and N must be empty AND block E must
*            not be captured by a straight moving vehicle) ]
*-------------------------------------------------------------------------
*
ENTER8R  BVARIABLE  (LS$LYTENSN*FNU$BLOKA*(LS$CLEARNSN+BV$CLEARNSN))+_
                    (LR$LYTENSN*FNU$BLOKA*FNU$BLOKK*FNU$BLOKL*_
                     FNU$BLOKH*FNU$BLOKN*LR$EBUBY5S)
*
*-------------------------------------------------------------------------
*                 L A N E 8    T R A V E L    T I M E S
*  Observed Average Travel Time - 3.660 seconds
*  Designated Travel Path       - A     K
*  Block Size Factor      (5.1) - 3     2.1
*  Travel Time Per Block (ms)   - 2153  1507
*-------------------------------------------------------------------------
*
         GENERATE   1000*&WEIBL(56.0592,0.63923,MX$SEED(&I,10))
*                                 A vehicle arrives in Lane 8
         QUEUE      STAT8R        Collect statistics for 8R vehicles
         SEIZE      FRONT8        Capture front end of Lane 8
         TEST E     BV$ENTER8R,1  Wait until the vehicle can enter the
*                                 intersection from Lane 8 to turn right
         TEST E     LS$LYTENSN,1,SKIP8R      If LYTENSN is red, skip
*                                            the next LOGIC Block
         LOGIC S    CLEARNSN      Intersection clearance was checked for
*                                 NS & SN traffic when light LYTENSN
*                                 just turned green
SKIP8R   SEIZE      BLOKA         Capture block A
         RELEASE    FRONT8        Free front end of Lane 8
         ADVANCE    2153          Travel on block A
         SEIZE      BLOKK         Capture block K
         RELEASE    BLOKA         Free block A
         ADVANCE    1507          Travel on block K
         RELEASE    BLOKK         Free block K
         DEPART     STAT8R        Record collected statistics
         TERMINATE  1             Exit the intersection
*
```

Figure 3.27 LANE8 Submodel

front of Lane 8 (**FRONT8**) to block K (**BLOKK**) to block A (**BLOKA**). If any one of these "facilities" is not available when requested with a **SEIZE** by the transaction, GPSS/H handles this "wait-until" condition at a low-level, hidden from the modeler. The modeler need not concern himself with determining "when" the facility becomes available. GPSS/H automatically makes the facility available to the transaction at the proper time. Execution of the **RELEASE** makes the facility available to the next waiting transaction, if any.

3.6.3 The EXPERIMENTAL CONTROL Submodel

Figure 3.28 gives the EXPERIMENTAL CONTROL submodel which resides between the **DO—ENDDO** looping construct, replicating the desired number of model execution runs, **NRUNS**. Each run includes a warmup (transient) period for the first 5000 transactions. After warmup, the **RESET** statement resets the statistical data, and model execution continues for 30000 additional transactions. The GPSS/H standard attribute **QT** (in conjunction with the designated statistics collection queues STAT1L, STAT2S, etc.) is specified for use to easily calculate the average waiting time per unit or transaction. Indexed on the run number, the performance measure arrays are loaded with these values at the GPSS/H **LET** assignment statements. The **RMULT** statement then specifies the starting seed values for the GPSS/H family of random number generators using the MX$SEED array. At the conclusion of each replication, the **CLEAR** statement zeroes the system clock, statistical counts, and other variables (except the MX$SEED array) in setting up for the next replication.

```
*=====================================================================
*        E X P E R I M E N T    C O N T R O L    S U B M O D E L
*=====================================================================
*
        DO          &I=1,&NRUNS    Replicate the sim. run &NRUNS times
*
           START       5000,NP     Warm up the model, produce No Print
           RESET                   Wipe out all statistics collected
*
           IF (&I<=2)OR(&I>=(&NRUNS-1))
                START   30000      Run for 30,000 vehicles in steady
*                                  state, produce standard GPSS/H output
           ELSE
                START   30000,NP   Run for 30,000 vehicles in steady
*                                  state, produce no standard output
           ENDIF
*
           LET         &AWTOV1L(&I)=QT$STAT1L/1000.
           LET         &AWTOV2S(&I)=QT$STAT2S/1000.
           LET         &AWTOV2R(&I)=QT$STAT2R/1000.
           LET         &AWTOV3L(&I)=QT$STAT3L/1000.
           LET         &AWTOV4S(&I)=QT$STAT4S/1000.
           LET         &AWTOV5S(&I)=QT$STAT5S/1000.
           LET         &AWTOV5R(&I)=QT$STAT5R/1000.
           LET         &AWTOV6L(&I)=QT$STAT6L/1000.
           LET         &AWTOV7S(&I)=QT$STAT7S/1000.
           LET         &AWTOV8R(&I)=QT$STAT8R/1000.
           LET         &AWTOV9L(&I)=QT$STAT9L/1000.
           LET         &AWTOV10S(&I)=QT$STAT10S/1000.
           LET         &AWTOV11S(&I)=QT$STAT11S/1000.
           LET         &AWTOV11R(&I)=QT$STAT11R/1000.
*
           IF &I<&NRUNS
              RMULT   ,MX$SEED(&I+1,1),MX$SEED(&I+1,2),MX$SEED(&I+1,3),_
                      MX$SEED(&I+1,6),MX$SEED(&I+1,7),MX$SEED(&I+1,11),_
                      MX$SEED(&I+1,12),MX$SEED(&I+1,13),MX$SEED(&I+1,14)
           ENDIF
*
           CLEAR       MX$SEED     Clear for the next replication
*
        ENDDO
*
```

Figure 3.28 EXPERIMENTAL CONTROL Submodel

3.6.4 The CI CONSTRUCTION Submodel

Following completion of all model replications, the CI CONSTRUCTION submodel in Figure 3.29 outputs the confidence interval calculations on the average waiting time data at each lane using the external FORTRAN routine CISUB. A typical output from the GPSS/H model is shown in Figure 3.30 is based on 30 replications of the GPSS/H model based on a transient period of 5000 transactions and a steady state period of 30000 transactions.

3.7 The OOP Application

SIMULA has traditionally been accepted as "the father of all object oriented languages" [Meyer 1987] and is therefore particulary suitable to demonstrate the OOP features discussed in Section 2.8. The SIMULA model which was covered in section 3.5 with respect to the PI CF will again be considered. Now, however, the model will be examined to determine how the OOP is utilized to assist the modeler in representing the model. We consider this SIMULA implementation for its use of encapsulation, inheritance, and activation/passivation, the principal features of the OOP. Each feature, as found in the SIMULA model, is discussed, in turn.

3.7.1 Encapsulation

The SIMULA class concept enables one to package data and its operations in a single coded structure. Thus, this package (the object) provides a means of data abstraction for a modeler. The SIMULA class has "had great influence on programming language design. Languages supporting the idea of data encapsulation (CLU, ALPHARD, MESA, CONCURRENT PASCAL) and so-called actor languages used by the artificial-intelligence

```
*-------------------------------------------------------------------
*          C.I.    C O N S T R U C T I O N    S U B M O D E L
*-------------------------------------------------------------------
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Turning Left from Lane 1'
         CALL        &CISUB(&AWTOV1L(1),&NRUNS,&TITLE)
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Traveling Straight from Lane 2'
         CALL        &CISUB(&AWTOV2S(1),&NRUNS,&TITLE)
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Turning Right from Lane 2'
         CALL        &CISUB(&AWTOV2R(1),&NRUNS,&TITLE)
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Turning Left from Lane 3'
         CALL        &CISUB(&AWTOV3L(1),&NRUNS,&TITLE)
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Traveling Straight from Lane 4'
         CALL        &CISUB(&AWTOV4S(1),&NRUNS,&TITLE)
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Traveling Straight from Lane 5'
         CALL        &CISUB(&AWTOV5S(1),&NRUNS,&TITLE)
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Turning Right from Lane 5'
         CALL        &CISUB(&AWTOV5R(1),&NRUNS,&TITLE)
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Turning Left from Lane 6'
         CALL        &CISUB(&AWTOV6L(1),&NRUNS,&TITLE)
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Traveling Straight from Lane 7'
         CALL        &CISUB(&AWTOV7S(1),&NRUNS,&TITLE)
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Turning Right from Lane 8'
         CALL        &CISUB(&AWTOV8R(1),&NRUNS,&TITLE)
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Turning Left from Lane 9'
         CALL        &CISUB(&AWTOV9L(1),&NRUNS,&TITLE)
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Traveling Straight from Lane 10'
         CALL        &CISUB(&AWTOV10S(1),&NRUNS,&TITLE)
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Traveling Straight from Lane 11'
         CALL        &CISUB(&AWTOV11S(1),&NRUNS,&TITLE)
*
         LET         &TITLE='Average Waiting Time of Vehi_
cles Turning Right from Lane 11'
         CALL        &CISUB(&AWTOV11R(1),&NRUNS,&TITLE)
*
         END                          Return control to Operating System
```

Figure 3.29 CONFIDENCE INTERVAL CONSTRUCTION Submodel

```
Average Waiting Time of Vehicles Turning Left from Lane 1

 NUMBER OF INDEPENDENT OBSERVATIONS =              30
 SAMPLE MEAN                        =       19.637207
 SAMPLE VARIANCE                    =        0.294181
 LIST OF INDEPENDENT OBSERVATIONS   :

      20.086243      19.345398      19.440018      19.385025      19.111282
      19.473602      19.900879      19.548843      20.199387      19.696686
      19.548157      19.550400      19.424332      19.573624      19.523773
      19.564423      19.775482      19.371231      19.022324      19.338715
      18.973450      20.215836      21.908401      19.974747      19.882477
      19.033157      19.168320      19.629761      19.548096      19.904144

 CONFIDENCE INTERVALS:

 ALFA LEVEL         0.10        0.05        0.025       0.01        0.005

 LOWER LIMIT       19.507      19.469      19.435      19.393      19.364
 UPPER LIMIT       19.767      19.805      19.840      19.881      19.910


...    ...    ...    ...    ...    ...    ...    ...    ...


Average Waiting Time of Vehicles Turning Right from Lane 8

 NUMBER OF INDEPENDENT OBSERVATIONS =              30
 SAMPLE MEAN                        =        5.052768
 SAMPLE VARIANCE                    =        0.221621
 LIST OF INDEPENDENT OBSERVATIONS   :

       4.843784       5.297090       4.691488       4.426093       4.497105
       5.011253       5.157214       4.238483       4.414792       4.243101
       5.252214       5.116699       5.320203       5.480015       5.349500
       5.504000       4.757237       4.790546       5.261300       5.224722
       5.345797       5.316929       5.435464       4.735308       4.651183
       5.039577       6.331944       5.893851       5.068257       4.888050

 CONFIDENCE INTERVALS:

 ALFA LEVEL         0.10        0.05        0.025       0.01        0.005

 LOWER LIMIT        4.940       4.907       4.877       4.841       4.816
 UPPER LIMIT        5.165       5.199       5.229       5.264       5.290


...    ...    ...    ...    ...    ...    ...    ...    ...
```

Figure 3.30 Output of Thirty Replications of the GPSS/H Model

community (PLASMA, ACT, SMALLTALK) are ultimately rooted in SIMULA " [Kreutzer 1986]. SIMULA has been a forerunner and model for the encapsulation feature found in languages supporting the OOP.

"The class concept is the central concept in the SIMULA programming language" [Palme 1976]. This is a strategic statement. The SIMULA class is "a block of data and of procedures operating on that data" [Palme 1976]. By partitioning a program and its data in such a way, the class provides structure, modularity, and more. "Processes comprised of like sets of activities are considered to belong to the same class. At any point in time, a number of such processes may exist in a system model, in varying stages of execution. Each is an instance or object of its class, uniquely identified among members of that class by certain attributes. The behavior of processes of the same class may be described by a single set of rules describing the activities of all processes from that class together with a set of attribute(s) values for each of the existing processes of that class" [Franta 1977]. Figure 3.31 provides an example of as class declaration and describes a light "direction". Figure 3.32 gives the class declaration of the light itself. Notice that within this object description that four "directions" are created with "new". Encapsulated within each direction are the procedural actions **setred** and **setgreen** which detail the actions that occur in accomplishing the transitions on light color between "red" and "green". The light controller's class declaration is shown in Figure 3.33. From the light controller perspective, the color transitions are simple to achieve. Taking advantage of the encapsulation of the color transitions, the light controller sends a communication "message" to the referenced light object which in turn executes the appropriate direction's "setred" or "setgreen" procedural actions. This is aptly demonstrated in the body of Figure 3.33 where the encapsulation feature is clearly utilized.

```
*************************************************************************
* DESCRIPTION: Class definition for light DIRECTION.
*
* ATTRIBUTES: Color of red or green and procedures to set color to red
*   or green.
* INPUT(S) : None
* OUTPUT(S): Change in color attribute when called.
*
* CALLS    : None
*
* CALLED BY: LIGHTCTRL process
*************************************************************************;

   class direction;
1     begin
2        boolean red, green;

3        procedure setred;                    !Sets light red ;
4        begin
5           red := true;
6           green := false;
7        end SETRED;

8        procedure setgreen;                  !Sets light green ;
9        begin
10          green := true;
11          red := false;
12       end SETGREEN;

13    end CLASS DIRECTION;
```

Figure 3.31 Class DIRECTION

```
*************************************************************************
* DESCRIPTION: Class definition for the LIGHT OBJECT.
*
* ATTRIBUTES: north, south, east, and west.  Represent the directions
*     of the light.  Each direction may hold "red" or "green".
* INPUT(S) : None
* OUTPUT(S): None
*
* CALLS    : None
*
* CALLED BY: The referenced light object is called by the LIGHTCTRL
*     Process
*************************************************************************;

class LIGHT;
1  begin
2     ref(direction) north, south, east, west;
3     north :- new direction;                !Set directions ;
4     south :- new direction;
5     east  :- new direction;
6     west  :- new direction;
7  end CLASS LIGHT;
```

Figure 3.32 Class LIGHT

```
*************************************************************************
* DESCRIPTION:  Class definition of the LIGHTCTRL (Light Controller)
*
* ATTRIBUTES: lite, the referenced LIGHT OBJECT being controlled.
*
* INPUT(S) :  lite, the LIGHT OBJECT
* OUTPUT(S):  "red" or "green" status is output to the directions of
*     the LIGHT OBJECT to simulate the light timing sequences.
* CALLS    :  No procedural calls, but remote access of the LIGHT
*     OBJECT is performed.
* CALLED BY:  Referenced by GENOBJECTS upon creation.
*************************************************************************;

process class LIGHTCTRL(lite);
1  ref(light)lite;
2  begin
3     while true do
4        begin
5           lite.north.setgreen;              !Set north and south green  ;
6           lite.south.setgreen;
7           lite.east.setred;                 !Set east and west red      ;
8           lite.west.setred;
9           reactivate this lightctrl delay 20 prior;
10          lite.north.setred;                ! Set north and south to red;
11          lite.south.setred;                !  for 1 sec clearance.     ;
12          reactivate this lightctrl delay 1 prior;
13          lite.west.setgreen;               ! Set West to green         ;
14          reactivate this lightctrl delay 13 prior;
15          lite.east.setgreen;               ! Now East set to green     ;
16          reactivate this lightctrl delay 16 prior;
17       end;                                 ! Now restart the cycle      ;
18    end CLASS LIGHTCTRL;
```

Figure 3.33 Class LIGHTCTRL

3.7.2 Inheritance

Concatenation is a binary operation on two SIMULA class declarations. The result of this operation is a new class, formed by "merging the attributes of both components (classes), and then combining their action (executable) statements" [Franta 1977]. For example, the class declaration of a model block object is shown in Figure 3.34. Now, Figure 3.35 illustrates the concatenation of the classes **BLOCK** and **BLOCKA**. As shown, "BLOCKA" is now called a subclass of "BLOCK". A class BLOCKA object is called a compound object. The result of this concatenation operation is that the instantiation of the BLOCKA object fully acquires (inherits) the attributes of the class BLOCK object as well as the additional attributes found in the class BLOCKA declaration. Hierarchical decomposition is then very easily done by the concatenation operation. Figures 3.19 and 3.20 in Section 3.5 also represent a clear example of the inheritance characteristic of the OOP. The attributes of the generic car (arrival and departure actions) were inherited by the individual cars which incorporated their own specific lane actions in their class declarations.

3.7.3 Activation and Passivation

A complete discussion of the activation and passivation features of the SIMULA model was provided in Section 3.5. Figure 3.20 in Section 3.5 (referred to above) includes the SIMULA "activate" and "passivate" primitives which enable the saving and restoration of an object's state between its periods of activity.

```
*************************************************************************
* DESCRIPTION: Class Definition of generic intersection BLOCK OBJECT
*
* ATTRIBUTES: busy      - boolean, occupied by transiting car
*             laneuser - lane identification of transiting car
*             turner    - boolean, car is a right turner
*             setbusy/free - set user variables and value of busy
*             free      - checks status of block attribute "busy"
*
* INPUT(S) : procedure setbusy is given pointer to transiting car
* OUTPUT(S): procedure free returns status of "busy" attribute
*
* CALLS     : None
*
* CALLED BY: Transiting car during transit routine (transitfm2, etc.)
*************************************************************************;

class BLOCK;
1  begin
2     boolean busy;
3     integer laneuser;
4     boolean turner;

5     procedure setbusy(user);
6        ref(car) user;
7        begin
8           laneuser := user.lane;          !Identify the user          ;
9           turner   := user.right;
10          busy := true;                   !Set busy = true            ;
11       end;

12    procedure setfree;
13       begin
14          laneuser := 0;                  !Reset user to nil values   ;
15          turner := false;
16          busy := false;                  !Set busy = false           ;
17       end;

18    boolean procedure free;
19       begin
20          if not busy then                !Check status of busy       ;
21             free := true
22          else
23             free := false
24       end FREE;

25  end CLASS BLOCK;
```

Figure 3.34 Class BLOCK

```
*************************************************************************
* DESCRIPTION: Class Definition for BLOCK A OBJECT.
*
* ATTRIBUTES:  procedure findtransit - determines transit time of car.
*
* INPUT(S) :   procedure findtransit is passed pointer to car object.
* OUTPUT(S):   procedure findtransit returns the transit time
*
* CALLS    :   None
*
* CALLED BY:   Transiting car from transit procedure (transitfm2,etc).
*              Also referenced by GENOBJECTS upon creation.
*************************************************************************;

BLOCK class BLOCKA;
1     begin
2        real procedure findtransit(vehicle);
3        ref(car)vehicle;
4        begin
5           inspect vehicle do
6              if lane = 8 then              !Return transit time based on;
7                 findtransit := 2.153;      ! lane id of car.            ;
8        end FINDTRANSIT;
9     end CLASS BLOCKA;
```

Figure 3.35 Class BLOCKA

3.8 The PGM Application

Applying the PGM to the TI was an extremely difficult task. No precedent exists for using the PGM in the domain of discrete event simulation. The examples in the literature all relate to the field of signal processing and provide little, if any, utility in guiding the modeling task since the domains of application are so totally different.

This section presents a discussion of the difficulties that were found in using the PGM and also the results of efforts to accomplish the construction of a representation of the TI with the PGM. Note that the application which resulted is not meant to be a complete specification of the TI. Only portions of the specification were completed, those portions that could provide sufficient detail to enable a rough evaluation of the PGM.

3.8.1 A Possible Approach to using PGM

The PGM is very effective in the context of signal processing applications, a computationally intensive domain. The reason for this is evident from the principal concepts around which the PGM is based. As discussed in Section 2.9, a connected network of nodes (the process graph) can be easily described by the PGM to represent a directed graph through which data flows from node to node. Each node represents a primitive function or computational instance. The necessary input for executing the function or computation enters the node from its input "queues" of data flow. The result of the node execution is placed on the output queue(s). The time of node execution is determined by the availability of the node input data at the input queue(s). Once this "scheduling criterion" is met, the node is executed. The number of data elements that are required at an input node (to enable the "firing" of the node) is determined by that input queue's *threshold* setting. This "firing" of the node is reminiscent of that found among nodes in Petri

nets [Peterson 1977].

At first glance, it would be reasonable to assume that such representational scheme would be perfect for the representation of object flow in a material-oriented manner, much like that discussed for the TF CF. As you remember, the activity-cycle diagrams (ACDs) in Section 3.3 also present a very similar viewpoint in depicting the direction of "entity" movement throughout a model instance as it moves from activity to activity. Therefore, a first-look (in a top-down fashion) at the TI might produce a graph instance, not unlike that in Figure 3.36 (showing vehicle flow from the Joint Lane and through the intersection from Lane 1 only). Vehicles are thought to queue up along the arcs of the nodes (the queues) and proceed through the intersection as each node executes, essentially pumping the vehicles along. Each node would represent a lane or block or special position within the intersection. However, the flow along the graph is determined by the timing of the node executions. These in turn are determined by more complex interactions than simply the presence of a vehicle at the input queue to a particular node. In addition, ECOS [Weitzman 1986] implementation of the PGM does not provide any effective timing mechanisms which might enable the delay of a node's output.

Figure 3.37 represents a more reasonable view to support object interactions within the realm of the PGM. Here the nodes (from Figure 3.36) are each transformed into sets of nodes as shown by the boxes, now labelled with the identifiers of the previous nodes. For example, Block I now becomes a "begin transit" node and an "end transit" node. The begin transit node can be fired by signal (CA_SIGNAL) when the conditions are met for that node execution. That is, the node will execute when the CA_SIGNAL is present, and when vehicles are present at the input of the Lane 1 Queue. When executing, also notice that a T_DELAY is output indicating the time delay or transit delay across Block I.

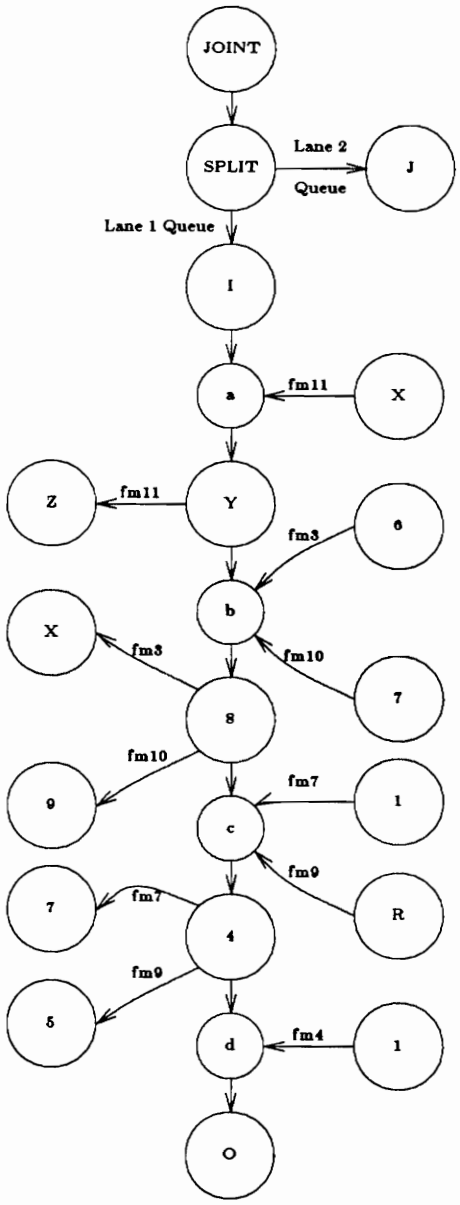


Figure 3.36 Initial Vehicle Flow

NOTES:

1. Queues a,b,c, and d place competing vehicles in FIFO order for next block.
2. fm_i = from lane i
3. At queue b, cars from lanes 3 and 10 compete for access to block 8.

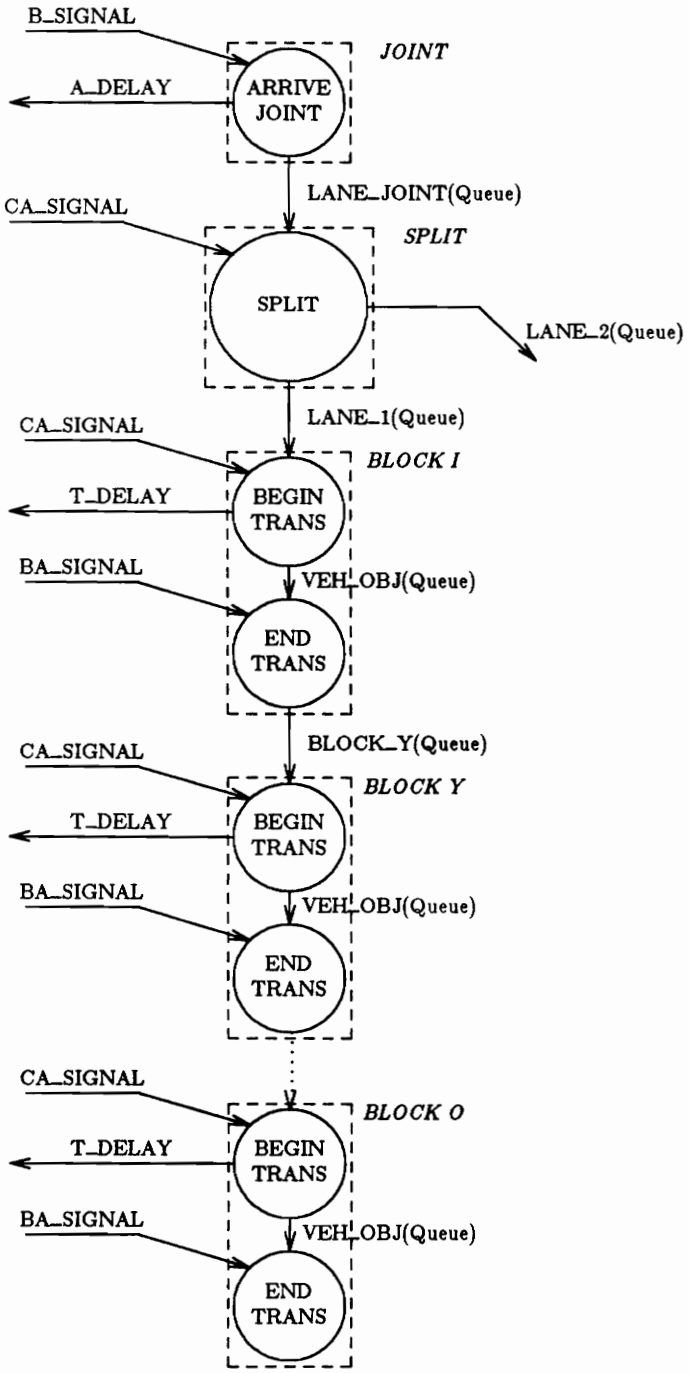


Figure 3.37 Improved Vehicle Flow

Similarly, the end transit node execution will occur when the firing signal (BA_SIGNAL) and vehicles in VEH_OBJ (Vehicle Object) Queue are present. Thus, we have each node representing an event, with firing signals controlling their execution, and the vehicle objects being pushed along their path. With this discussion as background, we can now classify nodes as B, BA, C, and CA nodes (taking the approach of the TPA) and fit this concept to the PGM. B and BA nodes are related to their "bound" activity counterparts. The B-nodes are used to determine if a B-activity is due. If due, the associated BA-node (the B-node's **A**ction equivalent) is executed. Similarly, the C-nodes represent the test-heads for the C-activities. If the C-node determines that its condition is satisfied, it signals its associated CA-node for execution.

Consider now the diagram of Figure 3.38. An additional graph structure can be attached to the B and C-nodes. Its data flow offers the executive control flow to manage the simulation. A time scan node receives the delays from nodes that schedule the "bound" activities. These delay times are scanned to determine their minimum value. This minimum is used to update the clock and to build a list of all the B-activities that are due. (Note that the delay times are initialized to enable the initial arrivals to the lanes or to 0. Each queue which contains a delay time has its threshold set at 0. Although the threshold setting remains constant for the life of the graph, other NEPs (Node Execution Parameters, like *read*, *consume*, and *produce*) may be variable, if so declared. The time scan node takes advantage of this variable NEP property to enable its selectivity of which delay time queues it will read and consume data on.) Once the time scan node execution is complete, the execution signal is passed to the B-nodes. These nodes use the list of B-activities that was compiled by the time scan node to determine if their associated BA-node should be activated. After each B-node is activated, a

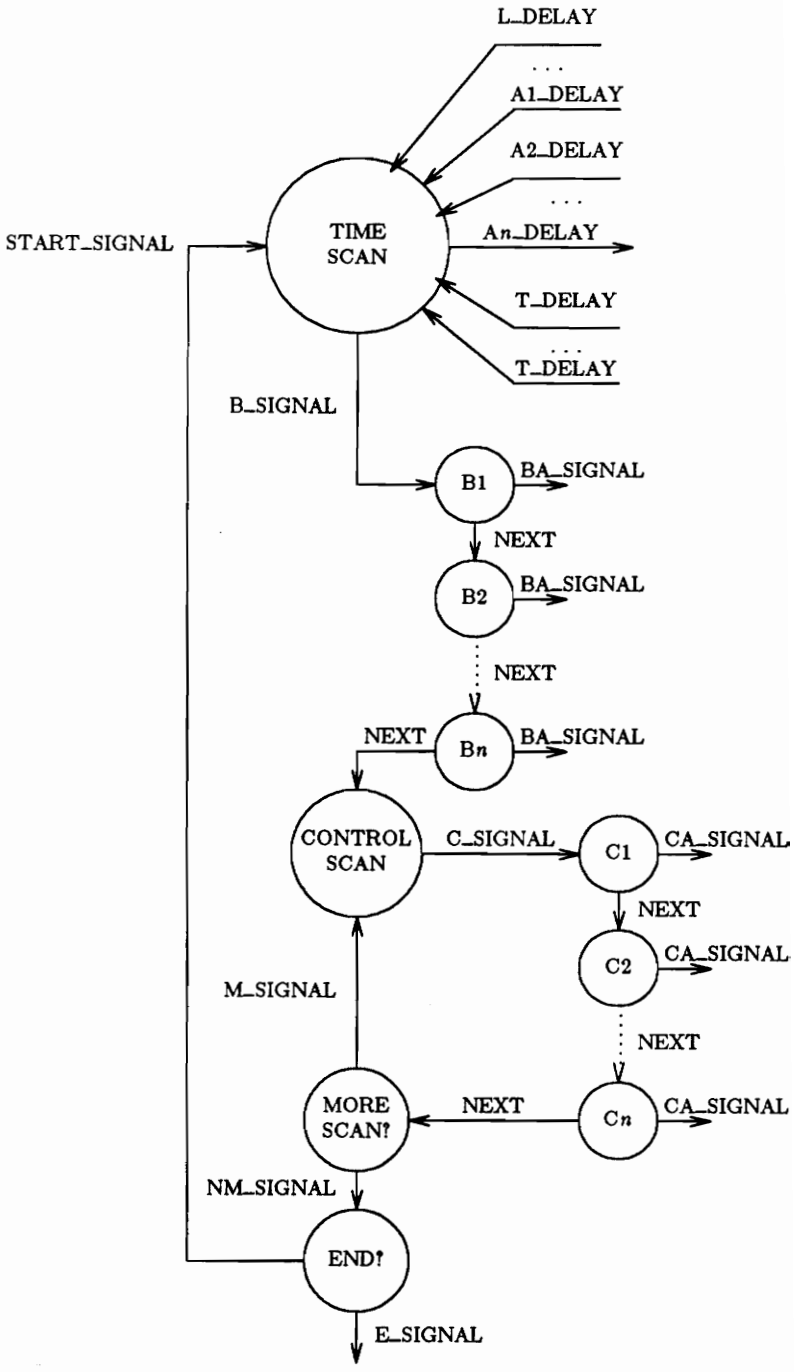


Figure 3.38 Executive Control Flow

similar scheme is followed by passing the execution signal to the C-nodes. The C-nodes, unlike the B-nodes, pass firing signals to their partner CA-nodes only upon satisfaction of their testing condition.

Following the format of the ECOS tutorial [Weitzman 1986], a sampling of the node descriptions is given for the TI. Figure 3.39 describes the Time Scan Node. Figures 3.40 through 3.42 show selected BA-nodes that represent the B- Activities. Figure 3.43 gives the C-node (testhead) for the CA-node "Begin Transit of Block Y" whose description is in Figure 3.44. Finally, Tables 3.1 and 3.2 provide the overall attribute declarations and queue descriptions, complete with NEP values.

3.8.2 Lessons Learned

The PGM can be effectively used to represent interactions among model components much like the ACD. However, no guidance is available to the modeler for model definition or specification. The modeler is strictly on his own in specifying the model dynamics and, in fact, must place his descriptions of the graph representation in the SPGN (Signal Processing Graph Notation) [Weitzman 1986], which is one, if not the only, available developmental notation for graph design. (The formats followed for the examples given in the preceding figures are strictly informal representations of the more complicated notation in the SPGN.) The conceptual representation is aided within the PGM by its ability to offer abstraction and modularity in that any node may be representing an underlying subgraph. In addition, PGM supports inheritance features in that nodes and/or families of nodes can be defined which can be used as templates for the creation of additional nodes. Thus, the newly created nodes inherit the features of their parent nodes or family.

PRIMITIVE NAME: TIME_SCAN

DESCRIPTION: The TIME_SCAN Node handles the update of system time by selecting the minimum time from all delay times. The node then schedules all B-Nodes which are due based upon the new system time. The TIME_SCAN Node implements the sequencing found in the TPA CF.

ALGORITHM:

Using the EVT_ARRAY, determine which delays among the DELAY ports to read/consume (Setting appropriate values for variable NEPs).
Load delay times into a temporary array.
Scan these times for the minimum time; set δ = minimum delay time.
$CLOCK = CLOCK + \delta$
Load BLIST with the due B-nodes for output.

PARAMETER LIST:

PRIMITIVE = TIME_SCAN
PRIM_IN= START_SIGNAL, CLOCK, EVT_ARRAY
PRIM_OUT= CLOCK, BLIST

INPUT TABLE			
Identifier	**Description**	**Mode**	**Range**
START_SIGNAL	Firing Signal	INT	{1}
CLOCK	System Time	FLOAT	R^+
EVT_ARRAY	Contains Scheduled B-nodes	INT ARRAY	{0(Not Sched),1(Sched)}

OUTPUT TABLE			
Identifier	**Description**	**Mode**	**Range**
CLOCK	System Time	FLOAT	R^+
BLIST	List of due B-nodes	INT ARRAY	Each Element {0,1}

Figure 3.39 Description of TIME_SCAN Node

PRIMITIVE NAME: NS_GREEN (BA-Node)

DESCRIPTION: The NS_GREEN Node performs the Action routine to change the North-South light to Green. In addition, the West and East lights are set to Red. The node is executed only when this routine becomes bound and scheduled in the EVT_ARRAY and when signalled by the BA_SIGNAL.

ALGORITHM:

```
NS_COLOR = 0   /* Set Green */
W_COLOR = 1    /* Set Red */
E_COLOR = 1
EVT_ARRAY [ NS_GREEN ] = 1  /* Schedule the Next Action Routine */
L_DELAY = 20.0   /* Set Delay Time Signal */
```

PARAMETER LIST:

PRIMITIVE = NS_GREEN
PRIM_IN= NS_COLOR, W_COLOR, E_COLOR, EVT_ARRAY, BA_SIGNAL
PRIM_OUT= NS_COLOR, W_COLOR, E_COLOR, EVT_ARRAY, L_DELAY

INPUT TABLE			
Identifier	**Description**	**Mode**	**Range**
NS_COLOR	North-South Color	INT	{0(Green),1(Red)}
W_COLOR	West Color	INT	{0(Green),1(Red)}
E_COLOR	East Color	INT	{0(Green),1(Red)}
EVT_ARRAY	Scheduled B-nodes	INT ARRAY	{0(Not Sched),1(Sched)}
BA_SIGNAL	Firing Signal	INT	{1}

OUTPUT TABLE			
Identifier	**Description**	**Mode**	**Range**
NS_COLOR	North-South Color	INT	{0(Green),1(Red)}
W_COLOR	West Color	INT	{0(Green),1(Red)}
E_COLOR	East Color	INT	{0(Green),1(Red)}
EVT_ARRAY	Scheduled B-nodes	INT ARRAY	{0(Not Sched),1(Sched)}
L_DELAY	Timing Delay	FLOAT	R^+

Figure 3.40 Description of NS_GREEN (BA Node)

PRIMITIVE NAME: ARR_LANE3 (BA-Node)

DESCRIPTION: The ARR_LANE3 Node performs the Action routine to accomplish a vehicle arrival to Lane 3. Bootstrapping the next arrival to Lane 3 is also accomplished. The node is executed only when this routine becomes bound and scheduled in the EVT_ARRAY and when signalled by the BA_SIGNAL.

ALGORITHM:

```
VEH_OBJ.LANE_ID = 3    /* Set Vehicle Attributes in Record */
VEH_OBJ.MOTION = N
VEH_OBJ.ARR_TIME = CLOCK
A_DELAY = GAMA | GSCALE, GSHAPE |    /* Set Delay Time to Next */
EVT_ARRAY | ARR_LANE3 | = 1    /* Arrival; Schedule B-Node */
LANE_3 = VEH_OBJ    /* Output Vehicle to Lane Queue */
```

PARAMETER LIST:

PRIMITIVE = ARR_LANE3
PRIM_IN= CLOCK, GSHAPE, GSCALE, BA_SIGNAL, EVT_ARRAY
PRIM_OUT= LANE_3, A_DELAY, EVT_ARRAY

INPUT TABLE			
Identifier	**Description**	**Mode**	**Range**
CLOCK	System Time	FLOAT	R^+
GSHAPE	Gamma Shape	FLOAT	R^+
GSCALE	Gamma Scale	FLOAT	R^+
BA_SIGNAL	Firing Signal	INT	{1}
EVT_ARRAY	Scheduled B-Nodes	INT ARRAY	{0(Not Sched),1(Sched)}

OUTPUT TABLE			
Identifier	**Description**	**Mode**	**Range**
LANE_3	New Vehicle	RECORD	LANE_ID={1,2,...,11} MOTION={N(Normal),R(Right)} ARR_TIME=R^+
A_DELAY	Next Arrival Time	FLOAT	R^+
EVT_ARRAY	Scheduled B-Nodes	INT ARRAY	{0(Not Sched),1(Sched)}

Figure 3.41 Description of ARR_LANE3 (BA Node)

PRIMITIVE NAME: END_TRANSIT_BLOCKY (BA-Node)

DESCRIPTION: The END_TRANSIT_BLOCKY Node performs the Action routine to complete the transit of a vehicle through Block 3. First, NEP Produce value is set for correct output queue; then the exiting vehicle is placed in that queue. The node is executed only when this routine becomes bound and scheduled in the EVT_ARRAY and when signalled by the BA_SIGNAL.

ALGORITHM:

NEP Calculation:

```
If VEH_OBJ.LANE_ID = 1 then
     PRODUCE for BLOCK_Z = 0   /* Set output for BLOCK_8 */
     PRODUCE for BLOCK_8 = 1
Else
     PRODUCE for BLOCK_8 = 0   /* Set output for BLOCK_Z */
     PRODUCE for BLOCK_Z = 1
```

Actions:

```
If VEH_OBJ.LANE_ID = 1 then
     BLOCK_8 = VEH_OBJ
Else
     BLOCK_Z = VEH_OBJ
```

PARAMETER LIST:

PRIMITIVE = END_TRANSIT_BLOCKY
PRIM_IN= VEH_OBJ, BA_SIGNAL
PRIM_OUT= BLOCK_8, BLOCK_Z

INPUT TABLE			
Identifier	**Description**	**Mode**	**Range**
VEH_OBJ	Transit Vehicle	RECORD	LANE_ID={1,2,...,11} MOTION={N(Normal),R(Right)} ARR_TIME=R^+
BA_SIGNAL	Firing Signal	INT	{1}

OUTPUT TABLE			
Identifier	**Description**	**Mode**	**Range**
BLOCK_8	Exiting Vehicle	RECORD	LANE_ID={1,2,...,11} MOTION={N(Normal),R(Right)} ARR_TIME=R^+
BLOCK_Z	Exiting Vehicle	RECORD	LANE_ID={1,2,...,11} MOTION={N(Normal),R(Right)} ARR_TIME=R^+

Figure 3.42 Description of END_TRANSIT_BLOCKY (BA Node)

PRIMITIVE NAME: BEGIN_TRANSIT_BLOCKY (C-Node)

DESCRIPTION: The BEGIN_TRANSIT_BLOCKY Node performs the test (testhead) for beginning transit of Block Y. First, NEP Produce value is set for correct output signal (i.e., whether CA_SIGNAL is turned on or off); then the testhead is performed.

ALGORITHM:

NEP CALCULATION:
If BLOCK_ARRAY [Y].QUEUECOUNT $>$ 0 and BLOCK_ARRAY [Y].STATUS = IDLE then
    PRODUCE for CA_SIGNAL = 1 /* Output turned on */
Else
    PRODUCE FOR CA_SIGNAL = 0 /* Output turned off */
Actions:
If BLOCK_ARRAY [Y].QUEUECOUNT $>$ 0 and BLOCK_ARRAY [Y].STATUS = IDLE then
    CA_SIGNAL = 1
NEXT = 1

PARAMETER LIST:
PRIMITIVE = BEGIN_TRANSIT_BLOCKY (C)
PRIM_IN= C_SIGNAL, BLOCK_ARRAY
PRIM_OUT= CA_SIGNAL, NEXT

INPUT TABLE			
Identifier	**Description**	**Mode**	**Range**
C_SIGNAL	Firing Signal	INT	{1}
BLOCK_ARRAY	Block Status Array	RECORD ARRAY	Each Record QUEUECOUNT=z^{+}_{0} STATUS={Idle,Busy} OCCLANE={1,2,...,11} OCCMOTION={N(Normal),R(Right)}

OUTPUT TABLE			
Identifier	**Description**	**Mode**	**Range**
CA_SIGNAL	Firing Signal	INT	{1}
NEXT	Firing Signal	INT	{1}

Figure 3.43 Description of BEGIN_TRANSIT_BLOCKY (C Node)

PRIMITIVE NAME: BEGIN_TRANSIT_BLOCKY (CA-Node)

DESCRIPTION: The BEGIN_TRANSIT_BLOCKY Node performs the Action routine for beginning transit of Block Y. The node is executed only when this routine is signalled by the CA_SIGNAL.

ALGORITHM:

```
VEH_OBJ = BLOCK_Y   /* Acquire the waiting vehicle */
BLOCK_ARRAY [Y].QUEUECOUNT = BLOCK_ARRAY [Y].QUEUECOUNT - 1
BLOCK_ARRAY [Y].STATUS = busy   /* Set Block Y attributes */
BLOCK_ARRAY [Y].OCCLANE = VEH_OBJ.LANE_ID  /* Set resident id */
BLOCK_ARRAY [Y].OCCMOTION = VEH_OBJ.MOTION  /* and motion */
If VEH_OBJ.LANE_ID = 1 then
     BLOCK_ARRAY [I].STATUS = idle  /*Free Block I */
     T_DELAY = SVCTIME (Y, 1, N)
Else   /* Vehicle is Lane 11 Car */.....
```

PARAMETER LIST:

PRIMITIVE = BEGIN_TRANSIT_BLOCKY (CA)
PRIM_IN= CA_SIGNAL, BLOCK_ARRAY, EVT_ARRAY, BLOCK_Y
PRIM_OUT= T_DELAY, EVT_ARRAY, VEH_OBJ

INPUT TABLE			
Identifier	**Description**	**Mode**	**Range**
CA_SIGNAL	Firing Signal	INT	{1}
BLOCK_ARRAY	Block Status Array	RECORD ARRAY	Each Record QUEUECOUNT=Z_0^+ STATUS={Idle,Busy}
EVT_ARRAY	Sched Array	INT ARRAY	{0(Not Sched),1(Sched)}
BLOCK_Y	Waiting Vehicle	RECORD	LANE_ID={1,2,...11} MOTION={N(Normal),R(Right)} ARR_TIME=R^+

OUTPUT TABLE			
Identifier	**Description**	**Mode**	**Range**
EVT_ARRAY	Sched Array	INT ARRAY	{0(Not Sched),1(Sched)}
VEH_OBJ	Transit Vehicle	RECORD	LANE_ID={1,2,...11} MOTION={N(Normal),R(Right)} ARR_TIME=R^+
T_DELAY	Transit Delay	FLOAT	R^+

Figure 3.44 Description of BEGIN_TRANSIT_BLOCKY (CA Node)

Table 3.1 PGM Variable Attribute Table

TYPE	NAME	MODE	DESCRIPTION
GV	**NS_COLOR**	INT [INIT to 1]	North-South Color
GV	**W_COLOR**	INT [INIT to 1]	West Color
GV	**E_COLOR**	INT [INIT to 1]	East Color
GV	**EVT_ARRAY**	INT ARRAY(50)§ [INIT to {0,0,0,...,0}]	B-Node Scheduling Array
GV	**BLIST**	INT ARRAY(50)§ [INIT to {0,0,0,...,0}]	Current B-Node Array (Due)
GV	**CLOCK**	FLOAT [INIT to 0.0]	System Time
GIP	**GSHAPE†**	FLOAT	GAMMA Shape Parameter
GIP	**GSCALE†**	FLOAT	GAMMA Scale Parameter
GV	**BLOCK_ARRAY**	RECORD_ARRAY(35)‡	Block Status Array
GV	**BSCAN**	INT	{0,1} Begin Scan Boolean
GV	**NDISS**	INT	Departures in Steady State
GV	**LOTP**	INT	Transient Period Counter
GIP	**LOSS**	INT	Length of Steady State Period

§ Size of array depends on number of B-Nodes.
† Shown for example purposes; other distribution constant information would be present.
‡ Size of array depends on number of Blocks.

Table 3.2 PGM Queue Attribute Table

Type	Queue Name	Mode	T§	R§	C§	O§	P§	Description
LOCAL	**START_SIGNAL**	INT [INIT to 1]	1	1	1	0	V	Start-up Signal
LOCAL	**L_DELAYs**	FIXED [INIT to 0.0]]	0	V	V	V	V	Light Delay
LOCAL	**A_DELAYs**	FIXED [INIT to 0.0]]	0	V	V	V	V	Interarrival Delay
LOCAL	**T_DELAYs**	FIXED [INIT to 0.0]]	0	V	V	V	V	Block Transit Delay
LOCAL	**B_SIGNAL**	INT	1	1	1	0	1	Start B-node Executions
LOCAL	**BA_SIGNALs**	INT	1	1	1	0	V	Start BA-node Execution
LOCAL	**C_SIGNAL**	INT	1	1	1	0	1	Start Condition Scan Execution
LOCAL	**CA_SIGNALs**	INT	1	1	1	0	V	Start CA-node Execution
LOCAL	**NEXT**	INT	1	1	1	0	1	Firing Signal to Next Node
LOCAL	**M_SIGNAL**	INT [INIT to 0]	1	1	1	0	V	Control for Control Scan
LOCAL	**NM_SIGNAL**	INT	1	V	V	V	V	Control for More Scan
LOCAL	**E_SIGNAL**	INT	1	V	V	V	V	Control for End Simulation
LOCAL	**VEH_OBJ**	RECORD	1	1	1	0	1	Vehicle Object Record
LOCAL	**LANE_*Xs***	RECORD	1	1	1	0	1	Lane Queues of Vehicle Records
LOCAL	**BLOCK_*Xs***	RECORD	1	1	1	0	1	Block Queues of Vehicle Records

§ Code for Table:

- T: Threshold
- R: Read
- C: Consume
- O: Offset
- P: Produce
- V: Variable

The PGM provides some useful tools during the design process, however provides little guidance in a "framework" sense. It is likely, too, that implementations of discrete event models using the data flow concepts of the PGM would be highly inefficient.

3.9 The ERA Application

One of the important contributions of the ER model [Chen 1976] was the introduction of a diagramming technique called the entity-relationship diagram which is extremely useful in designing databases. The ERA CF application to the TI is demonstrated using such an entity-relationship diagram. First, using the terminology which was introduced in Section 2.10, we describe the diagramming technique and its notational conventions which were developed by Chen [1976]. The TI is then described in terms of an entity-relationship diagram.

3.9.1 The Entity-Relationship Diagramming Technique

The three primitive concepts which form the basis of the ERA CF are the concepts of the *entity*, the *relationship*, and *value* [Dos Santos et al. 1980]. The entity-relationship diagram is used to depict the entities which are realized in a model, the relationships which exist between them, and the values of attributes which are associated with the entities or their determined relationships. The symbolic notation for an *entity set* is a box. *Relationship sets* are indicated by a diamond shape. Finally, *value sets* are notationally specified as circles. Normally, the attribute name that is attached to an entity is placed within the value set symbol [Hartson 1987]. Lines are used to interconnect the symbols in such a way as to display the existing role or attribute mappings between entity, relationship, and value sets. Figure 3.45 shows these symbols and their composition in a generic

diagram. Following these conventions, the symbols may take on the instance described by the diagram of Figure 3.46 in which an **student** entity set (with attributes of "student-id", "name", and "age") is related to the **course** entity set (with the attribute "course-name"). The relationship set **student_course** also includes an attribute, "grade". For simplification, all attribute names for a particular entity are enclosed within a single circle with the assumption that the domain of the value sets (from which the attribute values are derived) is logically implied from the associated attribute name. Also note that primary keys for a particular entity set are identified by underlining them [Hartson 1987].

Chen [1976] points out that a typical diagram may contain "several important characteristics about relationships in general". The diagram clarifies which entities enter into relationships. For example, a single relationship set may exist on one or more entity sets, or more than one relationship set can be defined on a group of entity sets. Also, the diagram distinguishes between one-to-one (1:1), one-to-many (1:n), and many-to-many (m:n) relationships. Normally, 1:1 relationships refer to the 1:1 mapping between an entity and its attributes, and 1:n relationships denote a hierarchical structure [Hartson 1987]. Figure 3.46 shows a many-to-many relationship since a single student may take many different courses and a single course may be taken by many different students.

3.9.2 An Entity-Relationship Diagram of the TI

In building an appropriate entity-relationship diagram for the TI, the four step procedure described in Section 2.10 was used. By following this procedure, the diagram structure of Figure 3.47 was determined.

- Step One — Identify entity and relationship sets of interest

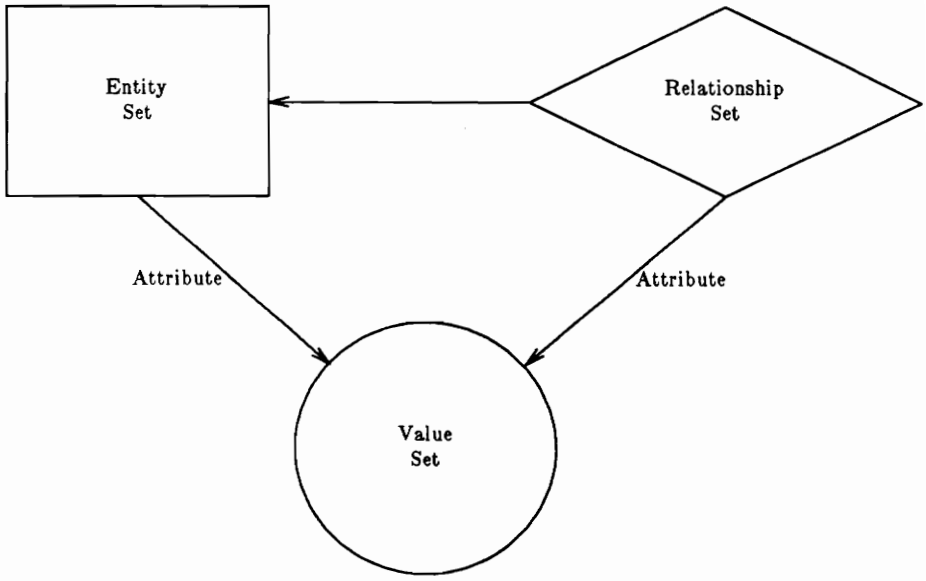


Figure 3.45 Generic Mappings in an Entity-Relationship Diagram [Dos Santos et al. 1980]

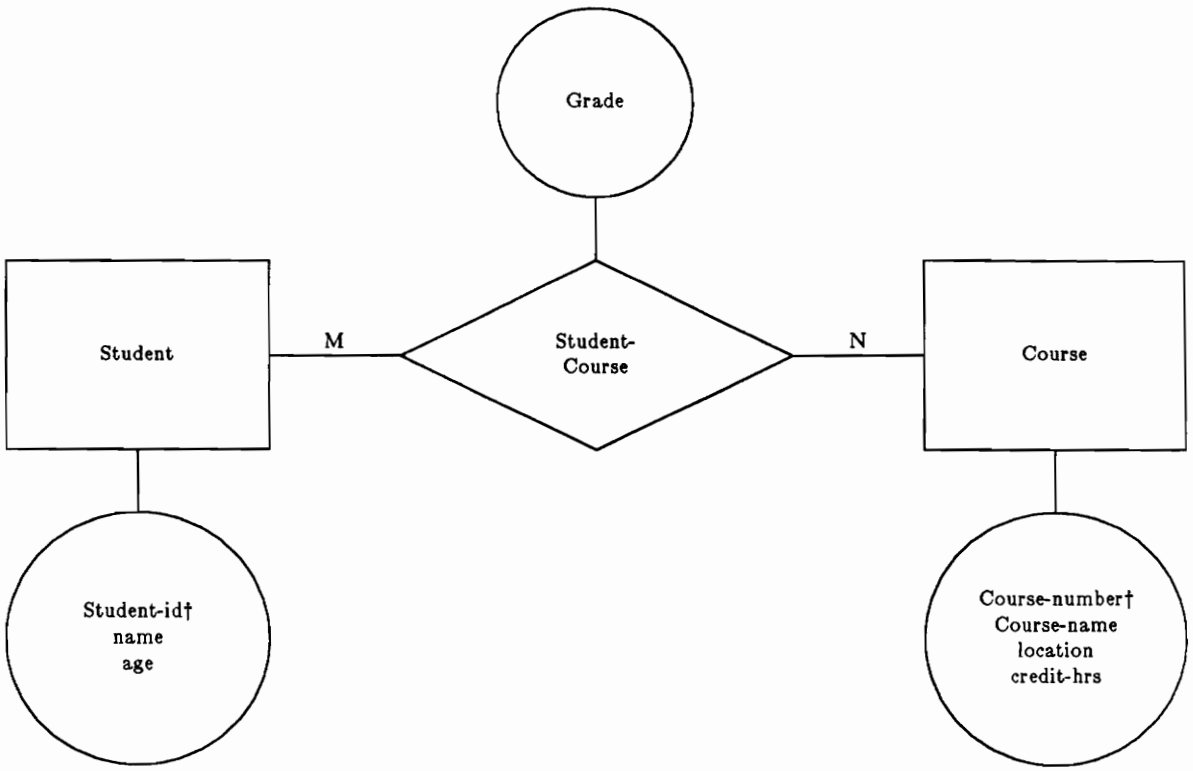


Figure 3.46 A Typical Entity-Relationship Diagram

† Primary Key

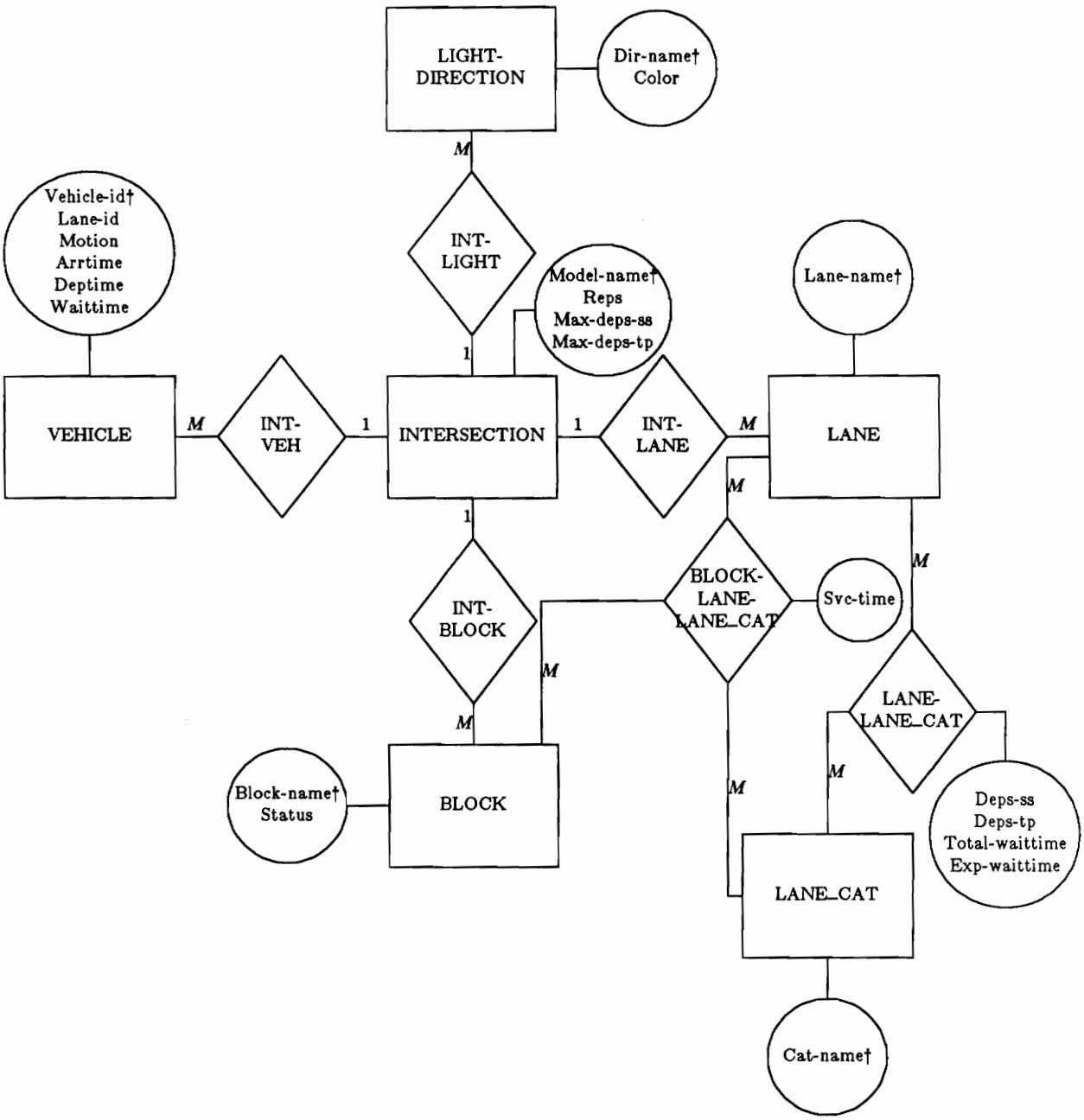


Figure 3.47 Entity-Relationship Diagram of the TI

† Primary Key

Entity sets necessary to represent the TI were identified as **VEHICLE**, **LIGHT_DIRECTION**, **BLOCK**, **LANE**, and **LANE_CAT** (Lane Category). In addition, an entity set to represent the entire model, **INTERSECTION** (which is composed of the above entity sets), was determined. At this point in the design process, two particularly important relationship sets were recognized. First, a **LANE-LANE_CAT** relationship set is needed to define the various paths a vehicle might travel dependent upon the lane and direction of movement. This relationship set is very important and serves as the basis for determining the desired performance measures. Secondly, block service times are related to the lane and to the lane category. The resulting relationship set, **BLOCK-LANE-LANE_CAT**, is determined from among these three indicated entity sets.

• Step Two — Identify semantic information

There is a 1:*m* mapping between the INTERSECTION entity set and each of the entity sets, VEHICLE, LIGHT_DIRECTION, BLOCK, and LANE. The LANE-LANE_CAT relationship is a *m:n* mapping. The BLOCK-LANE-LANE_CAT relationship is also a composed of many-to-many mappings. If not clear at this point, the semantic designations of the mappings as 1:*m* or *m:n* is clarified with the description of the attributes and their associated value sets in the next step.

• Step Three — Define value sets and attributes

Attributes for each entity set and the value sets of their attributes are shown in Table 3.3. Since each lane may be associated with two or fewer lane categories and each lane category may be related to any of eleven different lanes, the *m:n* relationship of the LANE-LANE_CAT relationship set is verified. Also notice that similar logic shows the

Table 3.3 ERA Entity Sets and Relationship Sets

Entity Set	Attribute	Value Set
INTERSECTION	Model_name	character string
	Reps	positive integers
	Max_deps_ss	positive integers
	Max_deps_tp	positive integers
VEHICLE	Vehicle_id	positive integers
	Lane_id	integer from 1..11
	Motion	R (right) or N (normal, straight)
	Arrtime	positive reals
	Deptime	positive reals
	Waittime	positive reals
LIGHT_DIRECTION	Dir_name	NS, E, or W
	Color	Red or Green
LANE	Lane_name	integer from 1..11
LANE_CAT	Cat_name	R (right) or N (normal, straight)
BLOCK	Block_name	character from A..Z or 1..9
	Status	idle or busy

Relationship Set	Attribute	Value Set
LANE-LANE_CAT	Deps_ss	positive integers
	Deps_tp	positive integers
	Total_waittime	positive reals
	Exp_waittime	positive reals
BLOCK-LANE-LANE_CAT	Svc_time	positive reals

validity of the many-to-many mappings for the BLOCK-LANE-LANE_CAT relationship set.

- Step Four — Organize data into relations, decide primary keys

The primary keys which uniquely specify the entity instances are indicated in Figure 3.47. The translation of the diagram into a relational data model [Date 1986] and the associated construction of relational tables is discussed by Chen [1976]. Hartson [1987] provides a good summary of the process. Essentially, simple entity relations are derived from the entity sets and their attributes. The one-to-many relationships do not become a relation. Instead, the primary key from the "one" entity becomes an attribute or part of the key of the "many" entity. The many-to-many relationships become a *linking* relation [Hartson 1987]. The primary key of this linking relation is derived from the keys of each of the "many" entities and remaining attributes are taken from the attributes of the relationship set, if any. Table 3.4 gives examples of such relations which are derived from the entity-relationship diagram of the TI model. The LIGHT_DIRECTION relation shows how the primary key of the INTERSECTION entity (with name TOMSCPF, for Toms Creek and Prices Fork Road) becomes part of LIGHT_DIRECTION's key in a 1:m derivation. BLOCK-LANE-LANE_CAT is a linking relation where the primary key is determined from the collaborating entities.

3.10 The EAS CF Application

Due to the claim stated in Section 2.11 that the EAS CF was derived, in part, from SIMSCRIPT, the SIMSCRIPT model (also used extensively in the ES CF application of the TI) is considered as a suitable example to illustrate the prominent features of the EAS CF. As a point of quick review, the EAS CF is primarily an approach which can be used

Table 3.4 Example Relations from the TI

LIGHT_DIRECTION		
Mod_name†	Dir_name†	Color
TOMSCPF	NS	Red
TOMSCPF	E	Red
TOMSCPF	W	Red

BLOCK-LANE-LANE_CAT			
Block_name†	Lane_name†	Cat_name†	Svc_time (ms)
A	8	R	2153
B	7	N	1071
C	6	N	1495
D	5	R	1577
E	5	N	933
E	5	R	1578
...	...	...	...

† Columns represent primary key

to represent the model "static" definition. Its central concepts include those of the *entity*, entity *attributes*, and collections of entities called *sets*. The SIMSCRIPT model of the TI includes such an approach for providing its static definition. This definition is realized in the formulation of the model **preamble**, portions of which are shown in Figure 3.48. Recall, however, that model "dynamics" are not easily represented but due to the "ordering" imposed on the sets, time and state relationships can be, in a limited sense.

3.10.1 Entities and Their Attributes

Every entity which is a component of the model is defined in the preamble following the key words **permanent entities** or **temporary entities**, whichever is more appropriate. Permanent entities in the TI, shown in lines 19-22 of Figure 3.48, include

- the *light*,
- the *blocks*, and
- the *lanes*.

The SIMSCRIPT standard convention for defining the entities is straightforward and includes an equally simple means of identifying the attributes of these entities. The light entity has the defined attributes of *color* for each specified direction. A block entity has the attibutes of *status*, *laneuser*, and *turner*. In the context of the SIMSCRIPT model, the status attribute indicates whether a block is "idle" (i.e., there is no transiting car occupying its physical space) or "busy" (a car is in the block). Laneuser and turner attributes both refer to characteristics of a car that is in the block, identifying the lane the car has come from and the car's direction of movement ("straight" or "right" turning). The turner attribute is actually a boolean that, when true, indicates the car is a right turner. The SIMSCRIPT representation does clearly show the associated attributes

```
preamble
1  event notices include
2     turn.ns.red and turn.ns.green and turn.west.green
3     and turn.east.green                        ''Event arguments include:
4     every departure has a out.vehicle          '' outgoing car
5     every arrival.blockd has a moving.car.d '' incoming car to block
6     every arrival.blockh has a moving.car.h
7     ...    ...    ...    ...    ...
8     every arrival.blockz has a moving.car.z
9     every arrival.block1 has a moving.car.1
10    ...    ...    ...    ...    ...
11    every arrival.block9 has a moving.car.9
12    every turning.left has a left.moving.car ''Car making turn
13    every enter has a in.vehicle              ''Car entering intersection
14    every arrival.joint has a incoming.car12 ''Car arriving lane
15    every arrival.lane1 has a incoming.car1
16    ...    ...    ...    ...    ...
17    every arrival.lane11 has a incoming.car11

18 normally, mode is integer
19 permanent entities
20    every light has a ns.color, a west.color and a east.color
21    every block has a status, a laneuser, a turner and owns a block.queue
22    every lane owns a lane.queue
23    ...    ...    ...    ...    ...
24 temporary entities
25    every car has an arrtime, a laneid,an id and a to.right
       and may belong to a block.queue
       and may belong to a lane.queue
       ...    ...    ...    ...    ...
26    define arrtime as a real variable
30    ...    ...    ...    ...    ...
31 end
```

Figure 3.48 The SIMSCRIPT Preamble with EAS CF Features

of these entities. Notice that there are no designated attributes for a lane entity.

For the TI, only the cars or vehicles are temporary and their definition is shown in lines 24-26 of the modified preamble of Figure 3.48. Each car is defined with attributes *arrtime, laneid, id,* and *to.right.* The arrtime attribute is the arrival time of the car. Laneid and to.right directly correspond to the block attributes of laneuser and turner. Laneid is the car's lane of origin and to.right is a boolean which logically indicates the car's direction of movement. The attribute id is a unique, sequential identification number given to each car arriving to the intersection.

Values held by all the attributes are of type integer with one exception. The "normally, mode is integer" statement in line 18 specifies this "normal" condition of value types. The exception, attribute arrtime, is defined in line 26, apart from the other attributes, to be of type real.

3.10.2 Set Ownership and Membership

The set definitions and designations of hierarchical structure for the EAS CF in the SIMSCRIPT model are included in the entity definitions discussed above. The key word **owns** following the specification of attributes, if any, describes those components which may be attached to or associated with specific entities. In Figure 3.48, queues are associated each block and each lane. Thus, a one-to-one relationship is indicated between these entities and their associated queues. Ownership of components or sets (as described above) is a way of showing hierarchical relationships in SIMSCRIPT. Membership in sets, like the queues mentioned above, is declared using the key words **may belong to** within an entity description. For example, in line 25, cars are defined as members of block and lane queues. Therefore, using set ownership and membership definitions,

relationships among model components is simplified.

3.11 The SM Application

Geoffrion [1988] provides a Structured Modeling Language (SML) which is a language for model definition. It is used to develop "text-oriented" and "table-oriented" model representations which are based on SM concepts. In this chapter, the SML is used to demonstrate the application of the SM CF for modeling the TI system. Following a brief description of the SML, a typical modular outline with a collection of elemental detail tables and an associated genus graph for a model of the TI are given. These three core aspects of a structured model representation support the earlier description of SM given in Section 2.13 and enable one to grasp the essential details of this CF. During the development of the SM application to the TI, there were difficulties with the representation of the dynamic relationships between model objects. The application is primarily geared toward a static representation and is, therefore, incomplete in that the model dynamics are not included.

3.11.1 Description of SML

As described above, the SML contains the necessary components which allow the construction of an indented list, text-oriented representation and also a table-oriented representation. The text-oriented form is described by the language through the use of a detailed notation which facilitates the development of a modular outline from which one can derive modular and generic information. The detailed data of the elemental structure is represented within the guidelines of SML's table-oriented notation, producing a set of elemental detail tables.

3.11.1.1 The Text-Oriented Notation

Central to the use of the text-oriented notation is the development of module and genus *paragraphs* for use as nodes in the indented list, rooted, textual tree. Each paragraph description (whether module or genus) is concluded with the reserved word ":.".

The syntax for the module paragraph [Geoffrion 1988] is

&MODULE_NAME :| *interpretation* :.

The ampersand denotes a node as a module paragraph and the ":|" is a reserved word which denotes the "end of the formal part" of the paragraph. In addition, the interpretation is an informal English description which introduces key words that amplify the meaning of the module name. The interpretation may reference other key words that have been previously introduced. Key words, when first introduced in an interpretation, are preceded and followed by the reserved word ":/" .

Genus paragraphs follow a similar but much more detailed notational syntax. The general form of the genus paragraph [Geoffrion 1988] is

GNAME [new index] [(generic calling sequence)] /**type**/ [index set statement] [:: domain statement] [: range statement] [; generic rule] :| [interpretation] :.

Square brackets indicate optional arguments. Specific forms for genus paragraphs differ according to the specified genus "type". Depending on the type, the number and applicability of options will vary. The type declaration is specified by one of the following reserved words:

/pe/ primitive entity

/ce/ compound entity

/a/ attribute

/va/ variable attribute

/f/ function

/t/ test

Each component of the genus paragraph fulfills a unique role. Each can be informally described as follows:

- The *new index* allows the specification of the individual elements within the genus GNAME by designating a symbolic index for each GNAME set member.
- The *generic calling sequence* indicates those genera (if any) which take part in the definition of GNAME.
- The *index set statement* "gives information about the population of the genus" GNAME.
- The *domain statement*, if used, specifies the "data type for the identifiers (names) used for the individual elements" of a genus using the new index option [Geoffrion 1988].
- The *range statement* gives the range set of allowable values for attribute genera.
- The *generic rule* indicates the rule of computation which determines the returned value of a function or test genus.
- The *interpretation* is exactly as described for the module paragraph.

Geoffrion [1988] gives an informal coverage (narrative in nature) that details the complete syntax requirements of each option in a set of informative appendices. Formal

coverage is also provided in an additional appendix which introduces a context-free grammar that describes the syntactic and lexical structure of the notation.

3.11.1.2 The Table-Oriented Notation

Proper use of the text-oriented notation will generate the "general structure" of a model representation. The SML also provides "a notation that is primarily table-based because tables can be an effective way to organize masses of element-level data ... effective both for people to grasp and for machines to process [Geoffrion 1988]. As noted previously, these tables are called the **elemental detail tables**.

The elements of each genus are usually put into the tables in a format that depends on genus type. Informally, the tables hold the naming conventions for genus elements, interpretations of the elements, and values, if appropriate. Normally, primitive and compound entity elements which are singletons are not represented in the tables since the genus paragraph contains all the necessary information. Other singleton elements (like attribute, function, and test singleton elements) become a single value cell. The specific details of how the tables are structured, loaded, and edited are given by Geoffrion [1988].

3.11.2 The Genus Graph

In developing the text and table-oriented representations for modeling the TI system, the basic element types were first identified. The TI system contains a number of lanes, blocks, vehicles, and a light as described in the CM definition in Section 3.1. Each collection of vehicles, lanes, and blocks were grouped into a primitive entity genus called VEHICLE, LANE, and BLOCK respectively. The light genus, LIGHT, is a singleton. Each of these primitive entity genera was further specified as follows:

- VEHICLE — Attribute genera (with value) and a function genus were defined for each vehicle. These include

  (1) LANE_ID (attribute), the lane number of the lane in which the vehicle is traveling,

  (2) ARR_TIME (attribute), a vehicle's arrival time to the intersection,

  (3) MOTION (attribute), a vehicle's direction of movement in its lane, whether straight or right turning,

  (4) DEP_TIME (attribute), a vehicle's departure time from the intersection, and

  (5) WAIT_TIME (function), the waiting time of a vehicle in the intersection.

- LANE — Additional compound entity genera were derived from the lane elements. Attribute genera are also defined on the LANE genus and its related compound entity genera.

  (1) JOINTLANE (compound entity) describes the relationship between lanes 1 and 2. Lanes 1 and 2 taken together are called the Joint Lane.

  (2) VIRT_LANE (compound entity), a set of conceptual lanes, was defined which represents the possible paths which vehicles may take, i.e., in lane 1 going straight, in lane 2 turning right, in lane 5 going straight, etc. An additional primitive entity genus, LANE_CAT (with categories denoting straight or right turning) was created to enable the definition of the virtual lanes.

  (3) CAPACITY (attribute) was defined for lanes 1 and 2 to hold the maximum vehicle capacity in these two lanes.

(4) DEPS_SS (attribute) and DEPS_TP (attribute) represent values which are maintained by the solver and are counters of the current numbers of vehicle departures in steady state or during the transient period. These current values can be compared against the maximum allowed values to fix the termination of execution.

- BLOCK — Each block has the following attribute genera:

  (1) STATUS (attribute), which indicates if the block is idle or busy (occupied by a vehicle), and

  (2) SVC_TIME (attribute), the vehicle transit time across a block which depends on the associated virtual lane of the transiting vehicle.

- LIGHT — The light determines a compound entity genus and its attribute genus.

  (1) DIRECTION (compound entity) represents the four directions (north, south, east, and west) of the light.

  (2) COLOR (attribute) is defined for each direction and may have red or green value. The color of each direction is updated by the solver and is used to control the flow of vehicles through the intersection.

The genus graph of Figure 3.49 shows the relationships which exist among these primitive entity genus types and their associated compound entity, attribute, and function genera. The genus graph also includes other function and attribute genera which are used by the solver to control model execution and to output useful statistical data. Attributes MAX_DEPS_SS, MAX_DEPS_TP, and REPS are used by the solver and represent the maximum number of departures (of vehicles) in steady state or during the transient period and also the number of replications that are desired. These values indicate the

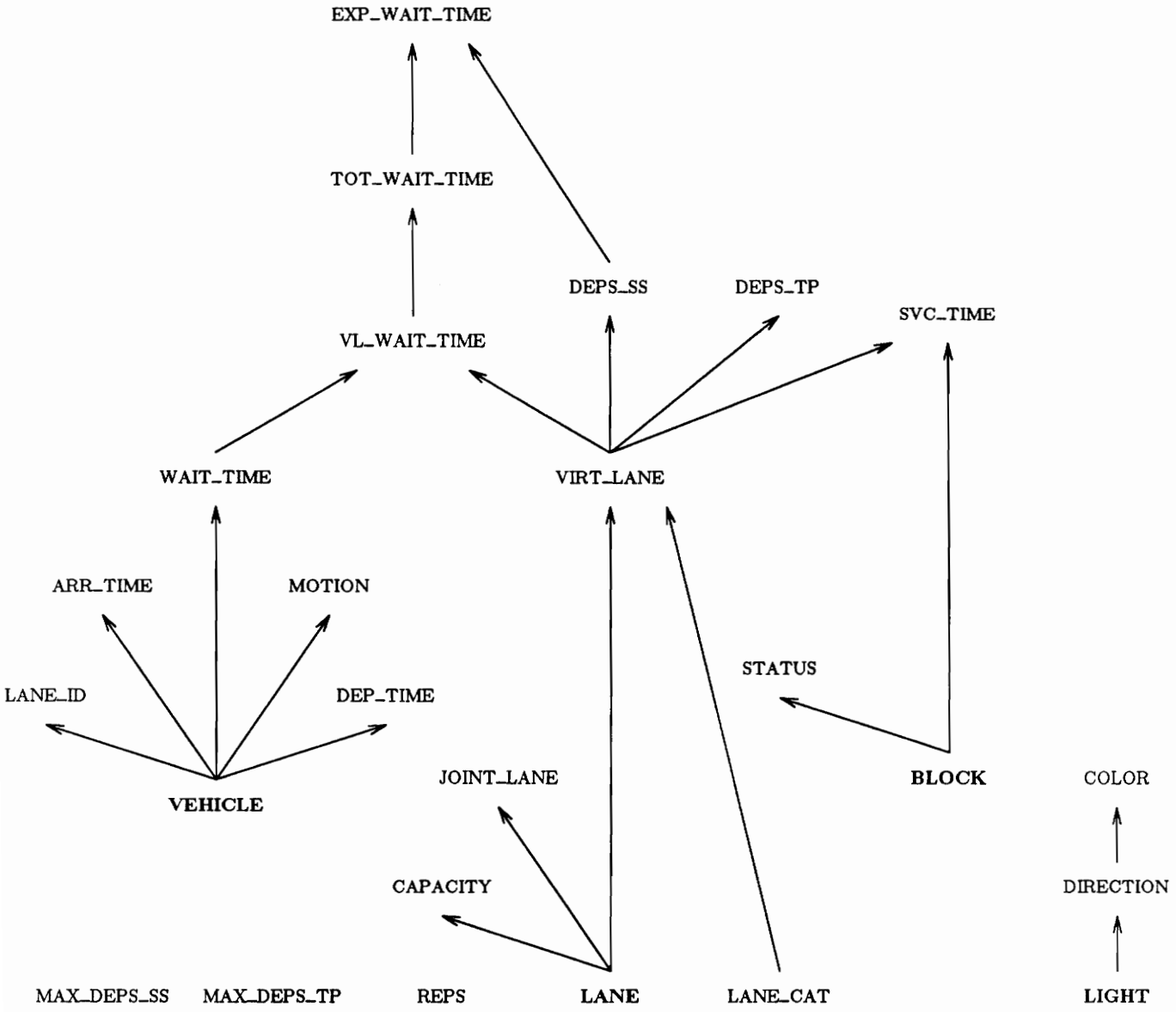


Figure 3.49 The TI Genus Graph

model execution termination conditions. Figure 3.49 also includes the statistical data that are derived from the above mentioned genera. The waiting time of each vehicle is associated with the vehicle's virtual lane by the function VL_WAIT_TIME, the virtual lane waiting time. The sum or total of all virtual lane waiting times in each virtual lane is calculated by the function TOT_WAIT_TIME. The function EXP_WAIT_TIME defines the expected waiting time for all transiting vehicles in a particular virtual lane.

3.11.3 SM Modular Outline and Elemental Detail Tables

The SM modular structure (conceptual in nature) is considered to be the collection of the basic primitive entity elements (&OBJECTS), the collective grouping of data and information about the vehicles (&VEH_DAT), the lanes (&LANE_DAT), the transit area (&TRANS_AREA_DAT, which includes block and light data), and statistics (&STAT_DAT). In addition, the modular structure includes overall model attributes (as shown in the genus graph) such as number of replications (REPS), etc., that assist in solver or executive control. Figure 3.50 shows a typical modular structure to the first sibling level. Figures 3.51 through 3.55 complete the modular structure by further developing the interior nodes (modules) of Figure 3.50 (i.e., &OBJECTS, &VEH_DAT, &LANE_DAT, &TRANS_AREA_DAT, and &STAT_DAT) to their descendant genera (leaf nodes).

The elemental detail tables provide the additional low-level information that cannot be represented by the genus and module paragraphs. The tables must be loaded with initial elemental information and are updated by the solver. The tables are structured and ordered so that the monotone property of the modular structure with no forward references is maintained. This eases the update operation or "editing" as Geoffrion [1988] calls

&TRAFFIC_INTERSECTION :| *There is a traffic :/INTERSECTION:/.* :.

&OBJECTS :| *The INTERSECTION is composed of key :/OBJECTS:/.* :.

&VEH_DAT :| *There is :/VEHICLE DATA:/.* :.

&LANE_DAT :| *There is :/LANE DATA:/.* :.

&TRANS_AREA_DAT :| *There is :/TRANSIT AREA DATA:/.* :.

MAX_DEPS_SS /a/ 1 : Int+ :| *There are a :/MAXIMUM NUMBER OF DEPARTURES IN STEADY STATE:/.* :.

MAX_DEPS_TP /a/ 1 : Int+ :| *There are a :/MAXIMUM NUMBER OF DEPARTURES IN TRANSIENT PERIOD:/.* :.

REPS /a/ 1 : Int+ :| *There are a designated number of :/REPLICATIONS:/.* :.

&STAT_DAT :| *There are :/STATISTICS:/.* :.:.

Figure 3.50 Overview of the Modular Structure (to First Sibling Level)

&OBJECTS :| *The INTERSECTION is composed of key, base :/OBJECTS:/.* :.

VEHICLE$_i$ /pe/ :: Int+ :| *There are :/VEHICLES:/ that transit the INTERSECTION.* :.

LANE$_j$ /pe/ Size {LANE} = 11 :: Int+ :| *There are :/LANES:/ in the INTERSECTION. Each :/LANE:/ serves as an entry point to the INTERSECTION.* :.

BLOCK$_k$ /pe/ Size {BLOCK} = 35 :: Int+ :| *There are :/BLOCKS:/ which serve as conceptual locations for VEHICLES in their transit of the INTERSECTION.* :.

LIGHT /pe/ Size {LIGHT} = 1 :| *There is a :/LIGHT:/ that controls the movement of the VEHICLES.* :.

Figure 3.51 Modular Structure of &OBJECTS

&VEH_DAT :| *:/VEHICLE DATA:/* :.

LANE_ID (VEHICLE$_i$) /a/ {VEHICLE} :Range {LANE} :| *Each VEHICLE arrives to the INTERSECTION in a certain LANE; Each VEHICLE has a :/LANE IDENTIFIER:/.* :.

ARR_TIME (VEHICLE$_i$) /a/ {VEHICLE} :R+ :| *Each VEHICLE has an :/ARRIVAL TIME:/ to the INTERSECTION.* :.

MOTION (VEHICLE$_i$) /a/ {VEHICLE} : "right","normal" :| *Each VEHICLE has an identifiable :/MOTION:/, and will either turn right on red (if possible) or will follow the normal flow of traffic in its LANE.* :.

DEP_TIME (VEHICLE$_i$) /a/ {VEHICLE} :R+ :|
Each VEHICLE has a :/DEPARTURE TIME:/ from the INTERSECTION. :.

WAIT_TIME (VEHICLE$_i$) /f/ {VEHICLE} ; DEP_TIME$_i$ - ARR_TIME$_i$:| *Each VEHICLE has a :/WAITING TIME:/.* :.

Figure 3.52 Modular Structure of &VEH_DAT

&LANE_DAT :| *:/LANE DATA:/* :.

JOINTLANE (LANE<1>, LANE<2>) /ce/ Size {JOINT} = 1 :| *There is a :/JOINTLANE:/ which is a composite LANE formed from LANES 1 and 2.* :.

LANE_CAT$_l$ /pe/ Size = 2 :: String :| *There are :/LANE CATEGORIES:/ which conceptually label a LANE as normal (straight) or as right turning.* :.

VIRT_LANE (LANE$_j$, LANE_CAT$_l$) /ce/ Select {TRANS_LANE} X {LANE_CAT} where j covers {LANE}, l covers {LANE_CAT} :| *There are :/VIRTUAL LANES:/ which are conceptual in nature and which coincide with the LANES but are additionally distinguished by a LANE CATEGORY.* :.

CAPACITY (LANE<1:2>) /a/ : Int+ :| *LANES 1 and 2 have an associated :/CAPACITY:/.* :.

DEPS_SS (VIRT_LANE$_{jl}$) /va/ {VIRT_LANE} : Int+ :| *There is a dynamically changing number of :/DEPARTURES IN STEADY STATE:/ which the solver maintains and corresponds to the number of VEHICLE departures in steady state from each VIRTUAL LANE.* :.

DEPS_TP (VIRT_LANE$_{jl}$) /va/ {VIRT_LANE} : Int+ :| *There is a dynamically changing number of :/DEPARTURES IN THE TRANSIENT PERIOD:/ which the solver maintains and corresponds to the number of VEHICLE departures in the transient period from each VIRTUAL LANE.* :.

Figure 3.53 Modular Structure of &LANE_DAT

&TRANS_AREA_DAT :| *:/TRANSIT AREA DATA:/* :.

&LIGHT_DAT :| *:/LIGHT DATA:/* :.

DIRECTION$_m$ (LIGHT) /ce/ Size {DIRECTION} = 4 :: Char 1 :| *Associated with the LIGHT are its :/DIRECTIONS:/ corresponding to the major points on the compass.* :.

COLOR (DIRECTION$_m$) /va/ {DIRECTION} : red, green :| *Each DIRECTION dynamically changes its :/COLOR:/.* :.

&BLOCK_DAT :| *:/BLOCK DATA:/* :.

STATUS (BLOCK$_k$) /va/ {BLOCK} : busy, idle :| *Each BLOCK has a :/STATUS:/ which is dependent on the presence of a transiting VEHICLE.* :.

SVC_TIME (BLOCK$_k$,VIRT_LANE$_{jo}$) /a/ Select {BLOCK} X {VIRT_LANE} : R+ :| *There is a :/SERVICE TIME:/ (Transit time) for each BLOCK which depends on the VIRTUAL LANE source of VEHICLES that may use the BLOCK.* :.

Figure 3.54 Modular Structure of &TRANS_AREA_DAT

&STAT_DAT :| *:/STATISTICAL DATA:/*

VL_WAIT_TIME (VIRT_LANE $_{jlm}$ WAIT_TIME$_i$) /f/ Select {VIRT_LANE} X {WAIT_TIME} : R+ ; WAIT_TIME$_i$:| *The modeler selects the index set such that each VIRTUAL LANE (first term of the Cartesian product) is paired with a WAITING TIME of any VEHICLE that is transiting that particular VIRTUAL LANE. The :/VIRTUAL LANE WAITING TIMES:/ are essentially a redesignation of the WAITING TIMES. That is, we have now associated each VEHICLE's WAITING TIME with a VIRTUAL LANE.* :.

TOT_WAIT_TIME (VL_WAIT_TIME $_{jli}$) /f/ {VIRT_LANE} : R+ ; SUM$_i$ VL_WAIT_TIME $_{jli}$:| *The :/TOTAL WAITING TIME:/ for a particular VIRTUAL LANE is computed by summing over all VIRTUAL LANE WAITING TIMES associated with that VIRTUAL LANE. That is, we have now provided for the calculation of the sum of all WAITING TIMES for VEHICLES transiting a particular VIRTUAL LANE.* :.

EXP_WAIT_TIME (TOT_WAIT_TIME $_{jl}$, DEPS_SS $_{jl}$) /f/ {VIRT_LANE} : R+ ; TOT_WAIT_TIME $_{jl}$ / DEPS_SS $_{jl}$:| *There is an :/EXPECTED WAITING TIME:/ for a VIRTUAL LANE which is calculated by dividing the TOTAL WAITING TIME for a given VIRTUAL LANE by the number of VEHICLES departing that VIRTUAL LANE during the steady state period.* :.

Figure 3.55 Modular Structure of &STAT_DAT

it. Tables 3.5 through 3.9 provide the complete elemental detail tables for the genera that are defined in this TI model. Table 3.5 shows a preliminary set of elemental detail tables for the primitive entities in the &OBJECTS module. The VEHICLE elemental detail table is further developed in Table 3.6 to include all vehicle attribute information. The BLOCK elemental detail table is also extended in Table 3.7 with the listing of the STATUS attribute. The TOT_WAIT_TIME and EXP_WAIT_TIME elemental detail (shown in Table 3.8) could be joined into a single table since both are indexed by virtual lanes. Finally, Table 3.9 shows the simple, single value elemental detail tables of the top level model attributes (e.g., REPS) that guide the solver.

3.12 The CS Application

This section describes the application of the CS to the TI. The figures of this section provide the complete specification of the TI and list the *interface, object, transition, function,* and *report* specification components of the CS. The syntax of the CS [Overstreet 1982; Overstreet and Nance 1985] is closely followed with some minor extensions which were necessary due to the complexity of the TI. These extensions deal with the means of object referencing and the creation of set objects. Overstreet and Nance [1985] recognized that complex models may require extensions in these areas beyond the original "abbreviated" treatment. In the subsections that follow, we discuss these extensions and explain, in detail, the components of this CS application and the corresponding figures.

Table 3.5 Preliminary Elemental Details of Base Objects

VEHICLE Elemental Detail	
VEHICLE	**INTERP**
1	Vehicle Number 1
2	Vehicle Number 2
3	Vehicle Number 3
...	...

LANE Elemental Detail	
LANE	**INTERP**
1	Lane 1
2	Lane 2
3	Lane 3
...	...
11	Lane 11

BLOCK Elemental Detail	
BLOCK	**INTERP**
1	Block A
2	Block B
3	Block C
...	...
26	Block Z
27	Block 1
28	Block 2
29	Block 3
...	...
35	Block 9

Table 3.6 Elemental Details of Vehicle and Lane Data

VEHICLE Elemental Detail						
VEHICLE	INTERP	LANE_ID†	ARR_TIME†	MOTION†	DEP_TIME†	WAIT_TIME†
1	Vehicle Number 1	2	1.5	normal	5.6	4.1
2	Vehicle Number 2	5	1.6	right	7.0	5.4
3	Vehicle Number 3	7	1.7	normal	8.1	6.4
...	...	...	...	...	...	...

LANE_CAT Elemental Detail	
LANE_CAT	INTERP
N	Normal, straight
R	Right turning

VIRT_LANE Elemental Detail				
LANE	LANE_CAT	INTERP	DEPS_SS†	DEPS_TP†
1	N	Virtual Lane 1N	1505	133
2	N	Virtual Lane 2N	1739	245
2	R	Virtual Lane 2R	648	71
3	N	Virtual Lane 3N	1321	211
4	N	Virtual Lane 4N	1276	309
5	N	Virtual Lane 5N	1894	266
5	R	Virtual Lane 5R	766	123
6	N	Virtual Lane 6N	962	117
7	N	Virtual Lane 7N	1278	225
8	R	Virtual Lane 8R	532	110
9	N	Virtual Lane 9N	1066	312
10	N	Virtual Lane 10N	1143	219
11	N	Virtual Lane 11N	1369	287
11	R	Virtual Lane 11R	592	107

CAPACITY Elemental Detail	
LANE	CAPACITY
1	5
2	5

† Data values in this column are shown for completeness, yet would be blank in the initial table and inserted by solver during execution.

Table 3.7 Elemental Details of Transit Area Data

DIRECTION Elemental Detail		
DIRECTION	**INTERP**	**COLOR†**
N	North	red
S	South	red
E	East	red
W	West	red

BLOCK Elemental Detail		
BLOCK	**INTERP**	**STATUS†**
1	Block A	idle
2	Block B	busy
3	Block C	idle
4	Block D	idle
...	...	...

SVC_TIME Elemental Detail			
BLOCK	**LANE**	**LANE_CAT**	**SVC_TIME(ms)**
1	8	R	2153
2	7	N	1071
3	6	N	1495
4	5	R	1577
5	5	N	933
5	5	R	1578
...	...	...	...

† Data values in this column are shown for completeness, yet would be blank in the initial table and inserted by solver during execution.

Table 3.8 Elemental Details of Statistical Data

VL_WAIT_TIME Elemental Detail			
LANE	LANE_CAT	WAIT_TIME†	VL_WAIT_TIME†
1	N	4.5	4.5
1	N	4.7	4.7
1	N	4.9	4.9
2	N	6.5	6.5
2	N	6.4	6.4
2	R	3.5	3.5
2	R	3.7	3.7
3	N	4.8	4.8
3	N	5.1	5.1
3	N	5.3	5.3
...	...	...	...

TOT_WAIT_TIME Elemental Detail Table		
LANE	LANE_CAT	TOT_WAIT_TIME†
1	N	14.1
2	N	12.9
2	R	7.2
3	N	15.2
...	...	...

EXP_WAIT_TIME Elemental Detail Table		
LANE	LANE_CAT	EXP_WAIT_TIME†
1	N	4.7
2	N	6.45
2	R	3.6
3	N	5.07
...	...	...

† Data values in this column are shown for completeness, yet would be blank in the initial table and inserted by solver during execution.

Table 3.9 Remaining Elemental Details

MAX_DEPS_SS
30000

MAX_DEPS_TP
5000

REPS
15

3.12.1 Syntax Extensions for Object Specification

The examples that have been given by Overstreet and Nance [1985] rely heavily on a Pascal-like syntax to represent the CS. One of the extensions developed for this application is the incorporation of the Pascal concept of enumerated types. Use of enumerated types enables the creation of and later identification of model objects with natural and more meaningful identifiers. As pointed out by Overstreet and Nance [1985], if "multiple instances of an object type can exist simultaneously, some mechanism must exist to uniquely identify individual instances of objects and object attributes when necessary." An object identifier which takes its values from an enumerated type definition is used to accomplish this. Figure 3.56 shows the definition of those enumerated types that are used in this application. Block identifiers may take values from within the range A to B9. This supports the block naming conventions that were specified in the CM definition of Section 3.1. Similarly, the range for identifiers of lane objects covers the values L1 (for lane 1) to JOINT. The range **dir_lane_range** defines the allowable identifier values of the special model object type, **dir_lane** (described in the next subsection). The implied meanings of these range values are, for example, normal or straight in lane 1 (N1) or right turning in lane 11 (R11).

Although Overstreet and Nance [1985] utilize a bracket index to reference object attributes where there are multiple instances of a single object type, Overstreet [1982] suggests that the dot notation like that used in SIMULA is appropriate. For example, to reference the attribute status of an instance of the block object type, the dot notation specifies that

block [i : A..B9]. status

```
{ Type Name                    Definition                                    }

block_range          =    (A, B, C, D, ..., Z, B1, B2, ..., B9);

lane_range           =    (L1, L2, ..., L11, JOINT);

dir_lane_range       =    (N1, N2, R2, N3, N4, N5, R5, N6,
                           N7, N8, N9, N10, N11, R11);
```

Figure 3.56 Use of Enumerated Types

is sufficient to pinpoint the "status" attribute. In contrast, the bracket index notation

status [i : A..B9]

accomplishes the same result. The dot notation, rather than the bracket notation, is used in this application and is preferred. The object to which the referenced attribute is attached is more clearly indicated.

3.12.2 Semantic Extensions for Object Specification

Overstreet and Nance [1985] state that for "complex models it may be necessary to regard some model objects as composed of both attributes and other model objects." The notion of a *set,* a model object that contains other model objects and which has attributes of its own, is another extension that is followed here. The object types of **block, lane,** and **dir_lane** are considered to be sets which contain an ordered collection of *vehicle* objects, like a queue. These sets are implicitly defined to have standard attributes and processing primitives (closely akin to those found in SIMSCRIPT) which include attributes of

- card (for cardinality) — in the standard sense, representing the number of objects in the set (nonnegative integer),
- first — which indicates the first object in the ordered collection (vehicle object),
- empty — indicating the empty condition of the set (Boolean),

and the primitives

- **put** — for inserting an object into the set in FIFO order, and
- **remove** — which removes a model object from the set.

Each of these sets qualifies as a *defined-set* in the CM terminology [Nance 1981a]. Briefly, the defined-set is a set object which contains objects and whose membership is determined dynamically by existing relationships among objects. On the other hand, the *primitive-set* contains objects with identical attributes and has a static membership. Use of the set objects enhances the specification process. For example, the set object **dir_lane** represents the collection of all vehicles which reside in a particular lane and which have the same direction of movement. (The SM application in Section 3.7 defined the virtual lane which corresponds in concept to this set object.) The attributes of the set members of a dir_lane object can be analyzed during the report phase of the specification to determine the desired performance measures. Nance [1988] notes that while perfectly reasonable in the specification of a model, the use of such a set during implementation would be impractical. Set member objects (vehicles) are temporary and would normally be destroyed on departure from the intersection. Yet, existence of the set object would require the necessary overhead to maintain and update its member objects for the entire duration of a single model execution.

3.12.3 Interface and Object Specifications

The interface and object specifications of the CS application to the TI are shown in Figures 3.57 and 3.58. The interface specification defines the input and output data. The input represents that information required to control the length of the simulation and the necessary probability distribution data. The output describes the performance measure requirements. The object specification includes the complete definition of all model objects and their attributes. Model object types in this application are *environment, light, vehicle,* and the set object types *block, lane,* and *dir_lane.*

```
Input:


loss            { Length of steady state period       } : positive integer;
lotp            { Length of transient period          } : positive integer;
gscale          { Gamma scale parameter               } : positive real;
gshape          { Gamma shape parameter               } : positive real;
wscale          { Weibull scale parameter             } : positive real;
wshape          { Weibull shape parameter             } : positive real;
mean            { Negative Exponential mean           } : positive real;
yvalues         { Y-axis values, 0..1, for use        }
                { in building a cumulative distribu-  }
                { tion function for random variate    }
                { generation by inverse transformation} : array of real;
xvalues         { X-axis values to associate with     }
                { above yvalues                       } : array of real;
svc_times       { Block service or transit times array}
                { which is 2-dim, Block X Lane_Dir    } : 2-dim array of real;

Output:
{ Expected or average waiting times for the set of    }
{ vehicles in each distinguishable lane path          } : nonnegative real
```

Figure 3.57 Traffic Intersection Interface Specification

```
{ Object     ::          Attribute              :          Type }

environment  ::          system_time            : positive real;
                         loss                   : positive integer;
                         lotp                   : nonnegative integer;
                         gscale                 : positive real;
                         gshape                 : positive real;
                         wscale                 : positive real;
                         wshape                 : positive real;
                         mean                   : positive real;
                         yvalues                : array of real;
                         xvalues                : array of real;
                         cleared_ns             : Boolean;
                         cleared_we             : Boolean;
                         ndiss                  : nonnegative integer;
                         svc_times              : 2-dim array of real;
                         l                      : nonnegative integer;

light        ::          ns_color               : (red, green);
                         west_color             : (red, green);
                         east_color             : (red, green);
                         ns_green               : time-based signal;
                         ns_red                 : time-based signal;
                         west_green             : time-based signal;
                         east_green             : time-based signal;
                         cleared_ns             : Boolean;
                         cleared_we             : Boolean;

vehicle      ::          l                      : nonnegative integer;
                         arr_time               : positive real;
                         wait_time              : positive real;
                         lane_id                : dir_lane_range;
                         wait_left              : Boolean;
                         departed               : Boolean;
                         id                     : positive integer;
                         arr_lane               : time-based signal;
                         end_trans              : time-based signal;
                         departure              : time-based signal;
                         delay                  : nonnegative real;

( Sets holding vehicle objects )
block        ::          status                 : (busy, idle);
                         occupant               : positive integer;
                         end_trans              : time-based signal;
                         departure              : time-based signal;

lane         ::          arr_lane               : time-based signal;

dir_lane     ::          tot_wait_time          : nonnegative real;
                         deps                   : nonnegative integer;
                         exp_wait_time          : nonnegative real;
```

Figure 3.58 Traffic Intersection Object Specifications

3.12.4 The Transition Specification

The transition specification provides the necessary details of the dynamic features, the time and state relationships, of the model and its objects. Included in the transition specification are the condition action pairs (CAPs) for initialization and termination, light changes, and for vehicle end block transits, lane arrivals, departures, and begin block transits. In addition, the specification of the special actions required by vehicles during branching in the joint lane (to lanes 1 or 2) and when making left turns before oncoming traffic are included. The transition specification is tailored to those actions of a lane 1 vehicle as it transits the intersection. For purposes of simplification, the specification of the condition action pairs (CAPs) for vehicles of other lanes is not included. However, a complete specification would be readily derivable from the given information.

A read operation retrieves the input data and the permanent model objects (light, blocks, lanes, and dir_lanes) are created during the initialization actions as shown in Figure 3.59. Model object attributes are also given initial values, including the environment object attributes of "cleared_ns", "cleared_we", "ndiss", and "l". "Cleared_ns" and "cleared_we" represent the Boolean condition of whether or not the intersection has been cleared for entry for the north-to-south traffic (or the south-to-north traffic) and for the west-to-east traffic, respectively. The first vehicle to enter the intersection for these directions must check the intersection clear (as specified in the CM definition of Section 3.1). Following vehicles do not have to make this check. "Ndiss" indicates the number of departures in steady state and "l" is an integer counter that indexes each vehicle object as it is created. Note that initialization also includes the "setting" of the initial light change and the initial arrivals to all lanes. Termination conditions are reached when the number

```
{ Initialization }
  INITIALIZATION:
     VAR  i : block_range;
          j : lane_range;
          k : dir_lane_range;
     READ (loss, lotp, gscale, gshape, wscale, wshape, mean, yvalues, xvalues);
     READ (svc_times);
     CREATE (light);
     ns_color := red;
     west_color := red;
     east_color := red;

  FOR i := A TO B9 DO
      CREATE ( block[i] );
      block [i].status := idle;
      block [i].occupant := 0;
      END FOR;

     cleared_ns := false;
     cleared_ns := false;
     ndiss := 0;
     l := 0;

  FOR j := L1 TO JOINT DO
      CREATE ( lane [j] );
      END FOR;

  FOR k := N1 TO R11 DO
      CREATE ( dir_lane [k] );
      dir_lane [k].tot_wait_time := 0;
      dir_lane [k].exp_wait_time := 0;
      dir_lane [k].deps := 0;
      END FOR

  SET ALARM (ns_green, 0);
  SET ALARM (arr_lane [JOINT], inv_trans (yvalues, xvalues));
  SET ALARM (arr_lane [L3], gamma (gscale, gshape));
  SET ALARM (arr_lane [L4], weibull (wscale, wshape));
  SET ALARM (arr_lane [L5], inv_trans (yvalues, xvalues));
  SET ALARM (arr_lane [L6], neg_exp (mean));
  SET ALARM (arr_lane [L7], weibull (wscale, wshape));
  SET ALARM (arr_lane [L8], weibull (wscale, wshape));
  SET ALARM (arr_lane [L9], inv_trans (yvalues, xvalues));
  SET ALARM (arr_lane [L10], inv_trans (yvalues, xvalues));
  SET ALARM (arr_lane [L11], inv_trans (yvalues, xvalues));

{ Termination }
  ndiss >= loss :
    STOP
```

Figure 3.59 Traffic Intersection Transition Specification (Initialization and Termination)

of departures in steady state equals the defined length of the steady state period.

The light change CAPs (Figure 3.60) dictate the light timing sequences of the various color changes for the light object. Each CAP includes within it the determination of the next light action with the SET ALARM primitive. Timing delays are in seconds.

CAPs which describe the actions to be taken by vehicles upon completion of a block transit, end block transit, are shown in Figure 3.61. Since only the actions for lane 1 vehicles are specified, only blocks I, O, Y, B4, and B8 are covered. These represent all the blocks in a lane 1 vehicle's path through the intersection. The scheduling of concurrent events is handled by the addition of the "NOT (ns_green ...)" portion of the condition expression for this action cluster. In effect, end block transit actions will be taken only when the alarm is due and there is no light change action due at the same time. This gives priority to the light change actions. A scan of the remainder of the transition specification shows how this feature, noted by Overstreet and Nance [1985], is accomplished within the CS. For simplification, some of the later concurrent scheduling and prioritizing conditions are not included (at the "NOT ...." statements, e.g., at Figure 3.64). Although the complete specification of priorities is not included, sufficient detail to describe the concept has been provided. The actions taken upon an end of transit of block O are essentially departure actions since block O is the last block in a lane 1 vehicle's transit path. The vehicle's waiting time is calculated, and a general departure action is scheduled immediately with the SET ALARM primitive.

Lane arrivals and departure actions are shown in Figure 3.62. Arrival to the JOINT lane results in actions which create the vehicle for the arrival, set the vehicle's initial attributes (including arrival time), and determine which lane (1 or 2) it will join. If the lane is to be 2, the direction of the vehicle (if to the right) is also determined

```
{ Light Changes }
{ North-south to green }
  WHEN ALARM (ns_green) :
     ns_color := green;
     west_color := red;
     east_color := red;
     cleared_we := false;
     SET ALARM (ns_red, system_time + 20)

{ North-south to red }
  WHEN ALRAM (ns_red) :
     ns_color := red;
     cleared_ns := false;
     SET ALARM (west_green, system_time + 1)

{ West to green }
  WHEN ALARM (west_green) :
     west_color := green;
     SET ALARM (east_green, system_time + 13)

{ East to green }
  WHEN ALARM (east_green) :
     east_color := green;
     SET ALARM (ns_green, system_time + 16)
```

Figure 3.60 Transition Specification (Light Changes)

```
{ End Block Transit }
  WHEN ALARM (end_trans [i : block_range] &
              NOT (ns_green OR ns_red OR west_green OR east_green)) :
    VAR veh_id : integer;
    veh_id := block [i].occupant;
    CASE i of
      A  : begin .. end;
      ...   ...   ...
      I  : begin
             put vehicle [veh_id] in block [Y];
           end;
      ...   ...   ...
      O  : begin
             CASE vehicle [veh_id].lane_id of
               1  : begin
                      block [O].occupant := 0;
                      block [O].status := idle;
                      vehicle [veh_id].wait_time :=
                        system_time - vehicle [veh_id].arr_time;
                      vehicle [veh_id].departed := true;
                    end;
               4  : begin .. end;
           SET ALARM (departure, 0);
           end;
      ...   ...   ...
      Y  : begin
             CASE vehicle [veh_id].lane_id of
               1  : put vehicle [veh_id] in block [B8];
              11  : put vehicle [veh_id] in block [Z];
           end;
      ...   ...   ...
      B4 : begin
             CASE vehicle [veh_id].lane_id of
               1  : put vehicle [veh_id] in block [O];
               7  : begin .. end;
               9  : begin .. end;
           end;
      ...   ...   ...
      B8 : begin
             CASE vehicle [veh_id].lane_id of
               1  : vehicle [veh_id].wait_left := true;
               3  : begin .. end;
              10  : begin .. end;
           end;
      B9 : begin .. end;
```

Figure 3.61 Transition Specification
(End Block Transits)

```
{ Lane Arrivals }
  WHEN ALARM (arr_lane [j : lane_range] &
              NOT (ns_green OR ns_red OR west_green OR east_green OR end_trans)):
    VAR  delay, draw_lane, draw_turn : positive real;
    CASE j of
      L1,L2 : ;
      L3    : begin .. end;
      ...   ...   ...
      L11   : begin .. end;
      JOINT : begin
                CREATE (vehicle [l]);
                vehicle [l].arr_time := system_time;
                vehicle [l].wait_left := false;
                vehicle [l].departed := false;
                vehicle [l].id := l;
                draw_lane := randm;
                if draw_lane <= .396 then
                   put vehicle [l] in dir_lane [N1]
                else begin
                   draw_turn := randm;
                   if draw_turn <= .213 then
                      put vehicle [l] in dir_lane [R2]
                   else
                      put vehicle [l] in dir_lane [N2];
                   end;
                put vehicle [l] in lane [JOINT];
                delay := inv_trans (yvalues, xvalues);
              end;
    SET ALARM (arr_lane [j], system_time + delay;
    l := l + 1

{ Departure }
  WHEN ALARM (departure &
              NOT (ns_green OR ns_red OR west_green OR east_green)) :
     IF lotp > 1 THEN
        lotp := lotp - 1
     ELSE begin
        IF lotp = 1 THEN begin
          lotp := 0;
          CLEAR dir_lane of vehicles where departed = true;
          end;
        ELSE
          ndiss := ndiss + 1;
        end;
```

Figure 3.62 Transition Specification
(Lane Arrivals and Departure)

probabilistically. The newly created vehicle is placed in the JOINT lane and the delay (inter-arrival time) to the next arrival to the JOINT lane is calculated. Arrival actions conclude with the setting of the alarm for the next arrival and incrementing the vehicle counter "l". Departure actions provide control over the duration of the simulation by managing the length of transient period ("lotp") and number of departures in steady state ("ndiss") attributes. Once the end of the transient period is reached, all vehicle objects that departed during the transient period are removed from the dir_lane sets, essentially resetting the model for purposes of statistical collection. From then on, the departure actions will update "ndiss" until termination conditions are reached.

Figures 3.63 and 3.64 include the CAP for vehicles to begin block transit. Each of the begin block transit CAPs represent contingent conditions. Upon entering the intersection at the first block (block I for lane 1 vehicles), the cleared_ns attribute (for the environment) is set to true for following vehicles. The vehicle is removed from its lane and the entered block is set to busy. Also, the block occupant attribute is set to the incoming vehicle. As a vehicle enters other blocks during its transit of the intersection, it is removed from the entered block's set and assigned to that block's space. This assignment results in that block's status being set to busy and the updating of the occupant attribute. When possible, the status and occupant attributes of blocks that have been "crossed" are reset (to idle and 0, respectively), "freeing" that block for occupancy by another vehicle object. End transit actions for the block being transited are determined and set at the conclusion of its begin transit action.

Figure 3.65 provides a description of the CAPs for the branching or splitting of a vehicle in the JOINT lane to lane 1 or to lane 2 and a description of the CAPs for accomplishing left turns. Note that these also contain contingent conditions.

```
{ Begin Block Transits }
...    ...    ...    ...
{ Block I }
  ((ns_color = green) & (block [I].status = idle) &
     (NOT lane [L1].empty) & (int_nsclear OR cleared_ns)) &
     NOT (ns_green OR ns_red OR west_green OR east_green OR
         end_trans OR arr_lane) :
     VAR tmpvehicle : vehicle;
     IF NOT cleared_ns THEN
        cleared_ns := true;
     tmpvehicle := lane [L1].first;
     remove tmpvehicle from lane [L1];
     block [I].status := busy;
     block [I].occupant := tmpvehicle.id
     SET ALARM (end_trans [I],
        system_time + trans_time (I,tmpvehicle.lane_id));
...    ...    ...    ...
{ Block O }
  NOT block [O].empty & block [O].status = idle & NOT ..... :
     VAR tmpvehicle : vehicle;
     tmpvehicle := block [O].first;
     remove tmpvehicle from block [O];
     block [O].status := busy;
     block [O].occupant := tmpvehicle.id
     CASE tmpvehicle.lane_id of
       N1   :begin
             block [B8].occupant := 0;
             block [B8].status := idle;
             block [B4].occupant := 0;
             block [B4].status := idle;
             SET ALARM (end_trans [O],
                system_time + trans_time (O, tmpvehicle.lane_id));
            end;
       N4   :begin .. end;
...    ...    ...    ...
{ Block Y }
  NOT block [Y].empty & block [Y].status = idle & NOT ..... :
     VAR tmpvehicle : vehicle;
     tmpvehicle := block [Y].first;
     remove tmpvehicle from block [Y];
     block [Y].status := busy;
     block [Y].occupant := tmpvehicle.id
     CASE tmpvehicle.lane_id of
       N1    :begin
             block [I].occupant := 0;
             block [I].status := idle;
             SET ALARM (end_trans [Y],
                system_time + trans_time (Y, tmpvehicle.lane_id));
            end;
       N11 :begin .. end;
...    ...    ...    ...
```

Figure 3.63 Transition Specification
(Begin Block Transits)

```
{ Begin Block Transits }
{ Block B4 }
  NOT block [B4].empty & block [B4].status = idle & NOT ..... :
     VAR tmpvehicle : vehicle;
     tmpvehicle := block [B4].first;
     remove tmpvehicle from block [B4];
     block [B4].status := busy;
     block [B4].occupant := tmpvehicle.id
     CASE tmpvehicle.lane_id of
       N1  :begin
            block [Y].occupant := 0;
            block [Y].status := idle;
            SET ALARM (end_trans [B4],
               system_time + trans_time (B4, tmpvehicle.lane_id));
           end;
       N7  :begin .. end;
       N9  :begin .. end;

...   ...   ...   ...
{ Block B8 }
  NOT block [B8].empty & block [B8].status = idle & NOT ..... :
     VAR tmpvehicle : vehicle;
     tmpvehicle := block [B8].first;
     remove tmpvehicle from block [B8];
     block [B8].status := busy;
     block [B8].occupant := tmpvehicle.id
     CASE tmpvehicle.lane_id of
       N1  :begin
            SET ALARM (end_trans [B8],
               system_time + trans_time (B8, tmpvehicle.lane_id));
           end;
       N3  :begin .. end;
       N10 :begin .. end;

...   ...   ...   ...
```

Figure 3.64 Transition Specification
(Begin Block Transits)

```
{ Splitting from Joint to Lanes 1 or 2 }
  (NOT lane [JOINT].empty) &
     ((( lane [JOINT].first.lane_id = N1) & (lane [L1].card <= 4)) OR
       (((lane [JOINT].first.lane_id = N2) OR (lane [JOINT].first.lane_id = R2))
       & (lane [L2].card <= 4))) & NOT ....
     VAR tmpvehicle : vehicle;
     tmpvehicle := lane [JOINT].first;
     remove tmpvehicle from lane [JOINT]
     CASE tmpvehicle.lane_id of
       1      : put tmpvehicle in lane [L1];
       2,2R   : put tmpvehicle in lane [L2];

{ Turning Left from Lane 1 }
   vehicle [block [B8].occupant].waiting = true & left1_ok & NOT ..... :
      vehicle [block [B8].occupant].waiting = false;
      put vehicle [block [B8].occupant] in block [B4];

{ Turning Left from Lane 3 }
...    ...    ...    ...

{ Turning Left from Lane 6 }
...    ...    ...    ...

{ Turning Left from Lane 9 }
...    ...    ...    ...
```

figure 3.65 Transition Specification
(Split and Turning)

3.12.5 The Function and Report Specifications

The final components of the CS are the function and report specifications, Figures 3.66 and 3.67. The function specification includes a brief description of the functions used by the CS. These functions are not fully defined for the sake of simplicity and include functions for the determination of the stochastic delays (inter-arrival times) for the initial and subsequent arrivals to the lanes. The function **trans_time** returns the transit or service time delay of a particular block based on the block's identifier and the identifier of its transiting vehicle. Also included are Boolean functions for checking clearance for making left turns and for entering the intersection. The report specification takes advantage of the utility of the dir_lane set. It specifies the performance measures as the summation of all wait_times of departed vehicles in a particular dir_lane object divided by the number of departed vehicles in that object.

3.13 The STA Application

The STA application develops both a static and dynamic specification of the TI using an AS approach. Figures 3.68 through 3.73 provide informal coverage in the manner suggested by Zeigler [1976] for an informative but non-technical description. Figures 3.74 through 3.83 complete the specification by recording its formal portions, in accordance with the DEVS (Discrete Event System Specification) formalism [Zeigler 1976, 1984a, 1984b, 1987; Concepcion and Zeigler 1988]. This section presents and discusses the use of the STA. A close review of the examples provided by the figures enables a deeper grasp of the technical details.

The examples in Zeigler's work [1976] do not clearly establish the means whereby temporary objects are created and destroyed. This application attempts to produce a

```
{ Function              Arguments                          Type }

weibull             (wscale         : real,
                     wshape         : real)          : positive real;
     begin .. end;

gamma               (gscale         : real,
                     gshape         : real)          : positive real;
     begin .. end;

neg_exp             (mean           : real)          : positive real;
     begin .. end;

inv_trans           (yvalues        : array of real,
                     xvalues        : array of real) : positive real;
     begin .. end;

randm                                                : positive real;
     begin .. end;

trans_time          (blockname      : block_range,
                     lanename       : lane_dir_range) : positive real;

left1_ok                                             : Boolean;
     begin .. end;

left3_ok                                             : Boolean;
     begin .. end;

left6_ok                                             : Boolean;
     begin .. end;

left9_ok                                             : Boolean;
     begin .. end;

int_nsclear                                          : Boolean;
     begin .. end;

int_westclear                                        : Boolean;
     begin .. end;
```

Figure 3.66 Traffic Intersection Function Specifications

```
{ Report Actions }
  WHEN end of simulation
    VAR i : dir_lane_range;
    FOR i := N1 TO R11 DO
        FOR every vehicle/ in dir_lane [i] where departed = true DO
            dir_lane [i].tot_wait_time :=
                 dir_lane [i].tot_wait_time + vehicle /.wait_time;
            dir_lane [i].deps := lane_dir [i].deps + 1;
            dir_lane [i].exp_wait_time :=
                 dir_lane [i].tot_wait_time / lane_dir [i].deps;
            END FOR;
    END FOR;
    FOR i := N1 TO R11 DO
       WRITE ("For Vehicles in Lane ", i,": ");
       WRITELN ("Expected Waiting Time is ", dir_lane [i].exp_wait_time);
       END FOR;
```

Figure 3.67 Traffic Intersection Report Specification

similar effect by assuming that the initial vehicle is available (created on initialization) with lane, direction, and arrival time descriptive variables set to null values. An arrival machine (one for each lane) operates on the vehicle counter, which is initialized to 1. The first machine to perform an arrival increments the vehicle counter to set up for the next vehicle. At each increment of the vehicle counter, a vehicle creation is implicitly understood to occur. Also note that the specification only includes transition descriptions of the blocks in a lane 1 vehicle path and that the specification is intended to support one replication.

3.13.1 The Informal Description

Following the general format that Zeigler [1976] suggests, an informal description is first presented of the model components (Figure 3.68), descriptive variables (Figures 3.69 and 3.70), and parameters (Figures 3.71 and 3.72). Model components are listed in Figure 3.68 as being *active* or *passive*. The active components are responsible for state changes among model components and thus "act on" other components and their descriptive variables. The passive components are not capable of this type of action and will change state only when "influenced" by an active component [Zeigler 1976].

The descriptive variables provide the state information of the model components. Each active component has at least two descriptive variables of the form "STATE OF" and "TIME LEFT IN STATE". The first is extremely important in specifying how a particular component "interacts" with the others. The latter corresponds to a countdown clock variable as mentioned in the overview of Section 2.15. Constants and functions which further help to define the model are called "parameters". The parameters of the TI application are listed and briefly described. For purposes of simplicity, the details of the

Components

Active:

LIGHTNS, *LIGHTW*, *LIGHTE*
ARRMACHINE·3, *ARRMACHINE*·4, . . . , *ARRMACHINE*·11, *ARRMACHINE*·*JNT*
BLOCK·*A*, *BLOCK*·*B*, . . . , *BLOCK*·*Z*
BLOCK·1, *BLOCK*·2, . . . , *BLOCK*·9
TURNER·1, *TURNER*·3, *TURNER*·6, *TURNER*·9
SPLITTER, *EXIT*, *TERM*

Passive:

VEHICLE·1, *VEHICLE*·2, · · ·
INTERSECTION, *MODEL*
LANE·*QUEUE*·1, *LANE*·*QUEUE*·2, . . . , *LANE*·*QUEUE*·11, *LANE*·*QUEUE*·*JNT*
WAIT·*QUEUE*·1, *WAIT*·*QUEUE*·3, *WAIT*·*QUEUE*·6, *WAIT*·*QUEUE*·9
STATISTICS·1, *STATISTICS*·2, . . . , *STATISTICS*·11*R*

Figure 3.68 Informal Description (Components)

Descriptive Variables

Describing $LIGHTx$ ($x = NS$ (*north—south*), W (*west*), E (*east*))
$STATE \cdot OF \cdot LIGHTx \rightarrow$ with range { 0(*green*) , 1(*red*)}, (s_x)
$x \cdot TIME \cdot LEFT \cdot IN \cdot STATE \rightarrow$ with range R_0^+, (σ_x)
$DELAY \rightarrow$ time delay determined by $LIGHTx$ and its state

Describing $ARRMACHINE \cdot y$ ($y = 1, \ldots, 11, JNT$)
$STATE \cdot OF \cdot ARRMACHINE \cdot y \rightarrow$ with range {0(*wait*), 1(*createvehicle*)}, (s_{My})
$My \cdot TIME \cdot LEFT \cdot IN \cdot STATE \rightarrow$ with range R_0^+, (σ_{My})
$INTERARRIVAL \cdot TIME \rightarrow$ rv determined by machine id, y

Describing $BLOCK \cdot k$ ($k = A, \ldots, Z, 1, \ldots, 9$)
$STATE \cdot OF \cdot BLOCK \cdot k \rightarrow$ with range {0(*idle*, *begin*), 1(*busy*, *end*) }, (s_{Bk})
$Bk \cdot TIME \cdot LEFT \cdot IN \cdot STATE \rightarrow$ with range $R_{0,\infty}^+$ (σ_{Bk})
$OCCUPANT \cdot OF \cdot BLOCK \cdot k \rightarrow$ with range {1,.., $VEH \cdot COUNTER$ }, (occ_{Bk})
$SVC \cdot TIME \rightarrow$ transit time delay determined by Block type, k,
and $l_{V(\,occ_{Bk})}$, and $d_{V(\,occ_{Bk})}$ of VEHICLE description

Describing $TURNER \cdot n$ ($n = 1,3,6,9$)
$STATE \cdot OF \cdot TURNER \cdot n \rightarrow$ with range {0(*wait*), 1(*turn*)}, (s_{Tn})
$Tn \cdot TIME \cdot LEFT \cdot IN \cdot STATE \rightarrow$ with range R_∞ , (σ_{Tn})

Describing *SPLITTER*
$STATE \cdot OF \cdot SPLITTER \rightarrow$ with range {0(*wait*), 1(*split*)}, (s_{SP})
$SP \cdot TIME \cdot LEFT \cdot IN \cdot STATE \rightarrow$ with range R_∞ , (σ_{SP})

Describing *EXIT*
$STATE \cdot OF \cdot EXIT \rightarrow$ with range {0(*wait*), 1(*depart*)}, (s_{EX})
$EXIT \cdot TIME \cdot LEFT \cdot IN \cdot STATE \rightarrow$ with range R_∞ , (σ_{EX})

Describing *TERM*
$STATE \cdot OF \cdot TERM \rightarrow$ with range {0(*wait*), 1(*terminate*)}, (s_{TERM})
$TERM \cdot TIME \cdot LEFT \cdot IN \cdot STATE \rightarrow$ with range R_∞ , (σ_{TERM})

Figure 3.69 Informal Description (Descriptive Variables, Active)

Descriptive Variables

Describing $VEHICLE \cdot j$ ($j = 1, \ldots, VEH \cdot COUNTER$)
(implying that $VEHICLE \cdot j$ exists for all $j \leq VEH \cdot COUNTER$)
$DIRECTION \cdot OF \cdot Vj \rightarrow$ with range {N(*normal*), R(*right*)}, (d_{Vj})
$LANE \cdot OF \cdot Vj \rightarrow$ with range {1, . . . ,11}, (l_{Vj})
$ARR \cdot TIME \cdot OF \cdot Vj \rightarrow$ with range R_0^+ , (A_{Vj})

Describing *MODEL*
$MODEL \rightarrow$ with range {0 (*transient*),1 (*transition*), 2 (*steady state*),
3 (*terminate*)} (s_{MOD})
$VEH \cdot COUNTER \rightarrow$ with range Z_0^+ , (s_{VC})
$NDISS \rightarrow$ with range Z_0^+ , (s_{NDISS})
$LOTP \rightarrow$ with range Z_0^+ , (s_{LOTP})

Describing *INTERSECTION*
$CLEARANCE \rightarrow$ with range {0(*clearedns*), 1(*clearedwe*)}, (s_{CL})

Describing $LANE \cdot QUEUE \cdot i$ ($i = 1, \ldots, 11, JNT$)
$LANE \cdot QUEUE \cdot i \rightarrow$ with range $\{1,2, \ldots, VEH \cdot COUNTER\}^*$, (s_{LQi})
(A sequence of vehicle components)

Describing $BLOCK \cdot QUEUE \cdot k$ ($k = A, \ldots, Z, 1, \ldots, 9$)
$BLOCK \cdot QUEUE \cdot k \rightarrow$ with range $\{1,2, \ldots, VEH \cdot COUNTER\}^*$, (s_{BQk})
(A sequence of vehicle components)

Describing $WAIT \cdot QUEUE \cdot n$ ($n = 1,3,6,9$)
$WAIT \cdot QUEUE \cdot n \rightarrow$ with range $\{1,2, \ldots, VEH \cdot COUNTER\}^*$, (s_{WQn})
(A sequence of vehicle components)

Describing $STATISTICS \cdot o$ ($o = 1,2,2R,3,4,5,5R,6,7,8,9,10,11,11R$)
$WAIT \cdot TIME \cdot o \rightarrow$ with range R_0^+ , (s_{WTo})

Figure 3.70 Informal Description (Descriptive Variables, Passive)

PARAMETER

LOSS → a constant indicating length of steady state period

INTERARRIVAL·TIME → a function that determines the time delay until the next vehicle arrival (using standard probability distributions and inverse transformation), based on arrival machine identifier and its interarrival seed; returns value in range R^+

DELAY → a function that determines the time delay until the next color change, based on light identifier and its state; returns value in range Z^+

SVC·TIME → a function that determines the transit time delay of vehicles transiting a particular block, based on the block and the transiting vehicle's lane and direction; returns value in range R^+

Figure 3.71 Parameters (Model Constants and Functions)

INTERARRIVAL·TIME (y, r_y) where y is *ARRMACHINE* identifier
and r_y is *INTERARRIVAL·SEED*
probability distributions for function with respect to y defined for:

$$y = \begin{cases} 3 & \rightarrow \textit{Gamma} \\ 4 & \rightarrow \textit{Weibull} \\ 6 & \rightarrow \textit{Neg\_exp} \\ 7 & \rightarrow \textit{Weibull} \\ 8 & \rightarrow \textit{Weibull} \\ \textit{else} & \rightarrow \textit{InverseTransform} \end{cases}$$

DELAY (x, s_x) where x is *LIGHT* identifier and s_x is state
and for the folloing arguments returns these values (in seconds):
NS, 0 → 30
NS, 1 → 20
W, 0 → 21
W, 1 → 29
E, 0 → 34
E, 1 → 16

SVC·TIME (k, $l_{V(\,occ_{Bk})}$, $d_{V(\,occ_{Bk})}$)
where k is a *BLOCK* identifier

STRING OPERATIONS:
top defined by *top* $(a_1\ a_2\ \cdots\ a_n) = a_1$ [Zeigler 1976]
rest defined by *rest* $(a_1\ a_2\ \cdots\ a_n) = (\ a_2\ \cdots\ a_n\ )$ [Zeigler 1976]
num defined by *num* $(a_1\ a_2\ \cdots\ a_n) = n$

Figure 3.72 Functions and String Operations

functions are not given. Only the essentials are listed. Note that the string operations shown in Figure 3.72 are extremely useful in the manipulation of the sets, a characteristic feature of the STA.

The model dynamics and "component interaction" are clarified with a plain English explanation in Figure 3.73. Model dynamics dictate that each component will transition from state to state as described. These transitions are translated into the formal transition functions which represent the bulk of the formal specification.

3.13.2 Beyond Informality in Time and State

The formalism of the STA makes it extremely difficult to use in practical applications like the TI example. Yet, where other CFs have failed (notably in their failure to adequately describe time and state relationships), the STA is effective. The time advance function is applied to the model state set to determine the next event time. This function essentially selects the minimum value found among the countdown variables. By association, one or more components which are due to change state are determined. [Note that if there is more than one component due to change state at the same instant, a prioritizing function (Zeigler [1976] uses SELECT) is used to break ties. Such tie-breaking rules are informally given in Figure 3.73.] Once "selected", an active component "executes" its formal transition function, and thereby produces the model state changes. The state changes are produced through the manipulation of descriptive variable values. The countdown clock variables must be accurately updated in order to maintain the proper time and state relationships.

Component Interaction

LIGHTx

0. Green for *DELAY·PERIOD*, then to state 1
1. Red for *DELAY·PERIOD*, returns to state 0

ARRMACHINE·y

0. Waits for *INTERARRIVAL·DELAY*
1. Creates vehicle, returns to state 0

BLOCK·k

0. Idle condition, waits for right conditions,
   (vehicle waiting to occupy and transit *BLOCK·y*)
   and then begins transit event
1. Busy condition, waits *SVC·TIME* and does end transit event
   and returns to state 0

TURNER·n

0. Waits for right conditions (oncoming traffic and blocks
   are clear) to do the turn
1. Does the turn, returns to state 0

SPLITTER

0. Waits for right conditions (vehicle is at the head of the
   Joint Lane and must branch to Lanes 1 or 2, and there is
   room in lanes 1 or 2(capacity of 5)) to do the split or branch
1. Does the split and returns to state 0

EXIT

0. Waits for right conditions (vehicle departure) to update
   departure information
1. Updates departure information, maintains model state, and returns
   to state 0

TERM

0. Waits for the model to enter state 3
1. Terminates model execution

TIE-BREAKING RULES

TERM in state 1 first, then any LIGHT, then EXIT in state 1, then BLOCKs in state 1, then ARRMACHINEs in state 1, then SPLITTER in state 1, then BLOCK in state 0, and finally, TURNERs in state 1.

Figure 3.73 Informal Description of Component Interactions

3.13.3 The Formal Specification

The structured formal specification [Zeigler 1976] is produced with a set description of the tie-breaking rules, and a clear indication of the "influencees" within the model. Informal coverage of the tie-breaking rules is sufficient for this discussion. The influencees are clearly distinguishable in the local transition functions, the principal components of the specification.

The local transition functions are given for each active component. As discussed in Zeigler [1976], these may be developed under the ES, AS, or PI CFs. The approach taken here closely follows the AS and the combined models approach described in Zeigler [1976].

For each component's transition function, we define a "condition routine" and an "activity routine". The condition routines are specified in such a manner that the occurrence of a state change for a particular component can be identified. When the condition is true and the component is selected (on the basis of analysis of the next imminent event data from the time advance function), the activity routine is executed. The use of negative-valued countdown variables enables the incorporation of the AS technique. When the condition is *determined*, the countdown variables can be exactly set so that they will reach zero at the precise instant at which the state change is to occur. For *contingent* conditions, however, the countdown variables are allowed to go negative so that when the condition routine becomes true, the component is an immediate candidate for selection.

Finally, all the condition routines can be incorporated into a list and "scanned". During each clock cycle, those component's condition routines which become true have their associated activity routines executed. The components whose condition routines are false during that same cycle have their countdown clock variables updated.

Figure 3.74 gives a simple example of the approach. The condition routine checks the state of the light, s_x. Therefore, whenever the state of any light is 0 or 1 (and the countdown variable produces that light component's selection), the activity routine is executed. The activity routine changes the state and resets the countdown variables. Table 3.10 shows one cycle of the state progressions of the lights. The local transitions which complete the specification are given in Figures 3.75 through 3.83. These are not discussed but follow the same concepts showed in Figure 3.74 (albeit at a more complex level).

3.11.4 Summary of the STA Application

Zeigler [1976] gives excellent and wide coverage of the DEVS in the context of the ES, AS, and PI CFs, with a few examples. The examples shed light on the practical aspects of the DEVS formalism. However, since many of these examples are incomplete (their completion is left as an "exercise" to the reader), the intricate details of the formalism are cloudy and a solid understanding of the approach is difficult to achieve. More recent work surrounding the development of a PC-based environment (PC-Scheme) should help to improve the modeler's ability to build model specifications based upon DEVS and the STA [Zeigler 1987].

STATE VARIABLES

For each *LIGHTx*, the following are *state* variables:
$STATE \cdot OF \cdot LIGHTx, (s_x)$
$x \cdot TIME \cdot LEFT \cdot IN \cdot STATE, (\sigma_x)$

CONDITION ROUTINE FOR *LIGHTx*

$C_x(s_x) \equiv$
1. $(s_x = 0)$ OR
2. $(s_x = 1)$

ACTIVITY ROUTINE FOR *LIGHTx*

$f_x(s_x, \sigma_x) = (s'_x, \sigma'_x)$

$s'_x = (s_x + 1) \ mod \ 2$

$\sigma'_x = DELAY(x, s_x)$

Figure 3.74 Local Transition for LIGHTx

Table 3.10 State Transitions for LIGHT$_x$

Descrip	Variable Value (at indicated times)				
Variable	t=0	t=20	t=21	t=34	t=50
s_{NS}	0	1	1	1	0
σ_{NS}	20	30	29	16	20
s_W	1	1	0	0	1
σ_W	21	1	29	16	21
s_E	1	1	1	0	1
σ_E	34	14	13	16	34

STATE VARIABLES:
For each *ARRMACHINE·y*, the following are *state* variables:
STATE·OF·ARRMACHINE·y(s_{My})
My·TIME·LEFT·IN·STATE(σ_{My})
INTERARRIVAL·TIME·SEED(r_{My})

CONDITION ROUTINE FOR *ARRMACHINE·y*:
$C_{My}(s_{My}) \equiv$
1. ($s_{My} = 0$) OR
2. ($s_{My} = 1$)

ACTIVITY ROUTINE FOR *ARRMACHINE·y*:
$f_{My}(s_{My}, \sigma_{My}, r_{My}, s_{LQy}, s_{VC}, l_{V(s_{VC})}, d_{V(s_{VC})}, A_{V(s_{VC})}) =$
$(s'_{My}, \sigma'_{My}, r'_{My}, s'_{LQy}, s'_{VC}, l'_{V(s_{VC})}, d'_{V(s_{VC})}, A'_{V(s_{VC})})$

$$s'_{My} = (s_{My} + 1) \bmod 2$$

$$\sigma'_{My} = \begin{cases} 0 & \text{if } s_{My} = 0 \\ INTERARRIVAL \cdot TIME\,(y, r_{My}) & otherwise \end{cases}$$

$$r'_{My} = \begin{cases} r_{My} & \text{if } s_{My} = 0 \\ \Gamma(r_{My}) & otherwise \end{cases}$$

$$s'_{LQy} = \begin{cases} s_{LQy} & \text{if } s_{My} = 0 \\ s_{LQy}\ s_{VC} & otherwise \end{cases}$$

$$s'_{VC} = \begin{cases} s_{VC} & \text{if } s_{My} = 0 \\ s_{VC} + 1 & otherwise \end{cases}$$

$$l'_{V(s_{VC})} = \begin{cases} l_{V(s_{VC})} & \text{if } s_{My} = 0 \\ y & otherwise \text{ if } y = 3, \ldots, 11 \\ 1,2 & otherwise\ (probalistic\ assign) \end{cases}$$

$$d'_{V(s_{VC})} = \begin{cases} d_{V(s_{VC})} & \text{if } s_{My} = 0 \\ N & otherwise \text{ if } y = 3,4,6,7,8,9,10 \\ R & otherwise\ (probalistic\ assign) \end{cases}$$

$$A'_{V(s_{VC})} = \begin{cases} A_{V(s_{VC})} & \text{if } s_{My} = 0 \\ CLOCK & otherwise \end{cases}$$

Figure 3.75 Local Transition for ARRMACHINEy

STATE VARIABLES:

$STATE{\cdot}OF{\cdot}BLOCK{\cdot}k(\ s_{Bk}\ )$

$Bk{\cdot}TIME{\cdot}LEFT{\cdot}IN{\cdot}STATE(\ \sigma_{Bk}\ )$

$OCCUPANT{\cdot}OF{\cdot}BLOCK{\cdot}k(\ occ_{Bk}\ )$

CONDITION ROUTINE FOR $BLOCK{\cdot}I$:

$C_{BI}(\ s_{NS}\ ,\ s_{LQI}\ ,\ s_{BI}\ ,\ s_{CL}\ ,\ s_{BL}\ ,\ s_{BH}\ ,\ s_{BN}\ ,\ s_{B1}\ ,\ s_{B2}\ ,$
$s_{B3}\ ,\ s_{BP}\ ,\ s_{BF}\ ,\ s_{BS}\ ,\ s_{BR}\ ,\ s_{B4}\ ,\ s_{B5}\ ,\ s_{B6}\ ,\ s_{BQ}\ ,\ s_{BG}\ ,$
$s_{BT}\ ,\ s_{BU}\ ,\ s_{B7}\ ,\ s_{B8}\ ,\ s_{B9}\ ,\ s_{By}\ ,\ s_{BZ}\ ,\ occ_{BE}\ ) \equiv$

1. $((\ s_{NS} = 0\ \ AND\ \ s_{BI} = 0)\ \ AND$
   $(s_{LQ1} \neq \Lambda\ \ AND\ \ ((s_{Bk}\ \text{for}\ all\ k\ monitored = 0\ \ AND\ \ l_{V(\ occ_{BE}\ )} \neq 5\ \ OR$
   $s_{CL} = 0))$, OR
2. $s_{BI} = 1$

ACTIVITY ROUTINE FOR $BLOCK{\cdot}I$:

$f_{BI}(\ s_{BI}\ ,\ \sigma_{BI}\ ,\ occ_{BI}\ ,\ s_{CL}\ ,\ s_{LQ1}\ ,\ s_{BQY}\ ) =$
$(\ s'_{BI}\ ,\ \sigma'_{BI}\ ,\ occ'_{BI}\ ,\ s'_{CL}\ ,\ s'_{LQ1}\ ,\ s'_{BQY}\ )$

$$s'_{BI} = 1$$

$$\sigma'_{BI} = \begin{cases} SVC{\cdot}TIME\ (\ I,\ l_{V(\ occ_{BI}\ )},\ d_{V(\ occ_{BI}\ )}\ ) & \text{if } s_{BI} = 0 \\ \infty & otherwise \end{cases}$$

$$occ'_{BI} = \begin{cases} top\ (\ s_{LQ1}\ ) & \text{if } s_{BI} = 0 \\ occ_{BI} & otherwise \end{cases}$$

$$s'_{CL} = \begin{cases} 0 & \text{if } s_{BI} = 0 \text{ and } s_{CL} = 1 \\ s_{CL} & otherwise \end{cases}$$

$$s'_{LQ1} = \begin{cases} rest\ (\ s_{LQ1}\ ) & \text{if } s_{BI} = 0 \\ s_{LQ1} & otherwise \end{cases}$$

$$s'_{BQY} = \begin{cases} s_{BQY} & \text{if } s_{BI} = 0 \\ s_{BQY}\ occ_{BI} & otherwise \end{cases}$$

Figure 3.76 Local Transition for BLOCKk, with BLOCK$\cdot I$

CONDITION ROUTINE FOR *BLOCK·Y*:

$C_{BY}(s_{BQY}, s_{BY}) \equiv$

1. $(s_{BQY} \neq \Lambda \ AND \ s_{BY} = 0)$, OR
2. $s_{BY} = 1$

ACTIVITY ROUTINE FOR *BLOCK·Y*:

$$f_{BY}(s_{BY}, \sigma_{BY}, occ_{BY}, s_{BI}, \sigma_{BI}, occ_{BI}, s_{BW}, \sigma_{BW}, occ_{BW}, s_{BQY}, s_{BQZ}, s_{BQ8}) =$$
$$(s'_{BY}, \sigma'_{BY}, occ'_{BY}, s'_{BI}, \sigma'_{BI}, occ'_{BI}, s'_{BW}, \sigma'_{BW}, occ'_{BW}, s'_{BQY}, s'_{BQZ}, s'_{BQ8})$$

$$s'_{BY} = 1$$

$$\sigma'_{BY} = \begin{cases} SVC \cdot TIME(Y, l_{V(occ_{BY})}, d_{V(occ_{BY})}) & \text{if } s_{BY} = 0 \\ \infty & otherwise \end{cases}$$

$$occ'_{BY} = \begin{cases} top(s_{BQY}) & \text{if } s_{BY} = 0 \\ occ_{BY} & otherwise \end{cases}$$

$$s'_{BI,BW} = \begin{cases} 0 & \text{if } s_{BY} = 0 \ AND \ l_{V(top(s_{BQY}))} = 1 \\ s_{BI,BW} & otherwise \end{cases}$$

$$\sigma'_{BI,BW} = \begin{cases} 0 & \text{if } s_{BY} = 0 \ AND \ l_{V(top(s_{BQY}))} = 1 \\ \sigma_{BI,BW} & otherwise \end{cases}$$

$$occ'_{BI,BW} = \begin{cases} 0 & \text{if } s_{BY} = 0 \ AND \ l_{V(top(s_{BQY}))} = 1 \\ occ_{BI,BW} & otherwise \end{cases}$$

$$s'_{BQY} = \begin{cases} rest(s_{BQY}) & \text{if } s_{BY} = 0 \\ s_{BQY} & otherwise \end{cases}$$

$$s'_{BQZ} = \begin{cases} s_{BQZ} & \text{if } s_{BY} = 0 \\ s_{BQZ} \, occ_{BY} & otherwise \end{cases}$$

$$s'_{BQ8} = \begin{cases} s_{BQ8} & \text{if } s_{BY} = 0 \\ s_{BQ8} \, occ_{BY} & otherwise \end{cases}$$

Figure 3.77 Local Transition for BLOCK·*Y*

CONDITION ROUTINE FOR *BLOCK*·8:

$C_{B8}(s_{BQ8}, s_{B8}) \equiv$
1. $s_{BQ8} \neq \Lambda$ *AND* $s_{B8} = 0$), OR
2. $s_{B8} = 1$

ACTIVITY ROUTINE FOR *BLOCK*·8:

$f_{B8}(s_{B8}, \sigma_{B8}, occ_{B8}, s_{WQ1}, s_{BQ8}, \ldots) =$
$(s'_{B8}, \sigma'_{B8}, occ'_{B8}, s'_{WQ1}, s'_{BQ8}, \ldots)$

$$s'_{B8} = 1$$

$$\sigma'_{B8} = \begin{cases} SVC \cdot TIME(8, l_{V(occ_{B8})}, d_{V(occ_{B8})}) & \text{if } s_{B8} = 0 \\ \infty & otherwise \end{cases}$$

$$occ'_{B8} = \begin{cases} top(s_{BQ8}) & \text{if } s_{B8} = 0 \\ occ_{B8} & otherwise \end{cases}$$

$$s'_{WQ1} = \begin{cases} s_{WQ1} & \text{if } s_{B8} = 0 \\ s_{WQ1}\, occ_{B8} & otherwise \end{cases}$$

$$s'_{BQ8} = \begin{cases} rest(s_{BQ8}) & \text{if } s_{B8} = 0 \\ s_{BQ8} & otherwise \end{cases}$$

. . . . . .

Figure 3.78 Local Transition for BLOCK·8

CONDITION ROUTINE FOR *BLOCK*·4:

$C_{B4}(s_{BQ4}, s_{B4}) \equiv$
1. $s_{BQ4} \neq \Lambda$ *AND* $s_{B4} = 0$), OR
2. $s_{B4} = 1$

$f_{B4}(s_{B4}, \sigma_{B4}, occ_{B4}, s_{BY}, \sigma_{BY}, occ_{BY}, s_{BQO}, s_{BQ4}, \ldots) =$
$(s'_{B4}, \sigma'_{B4}, occ'_{B4}, s'_{BY}, \sigma'_{BY}, occ'_{BY}, s'_{BQO}, s'_{BQ4}, \ldots)$

$$s'_{B4} = 1$$

$$\sigma'_{B4} = \begin{cases} SVC \cdot TIME(4, l_{V(top(s_{BQ4}))}, d_{V(top(s_{BQ4}))}) & \text{if } s_{B4} = 0 \\ \infty & otherwise \end{cases}$$

$$occ'_{B4} = \begin{cases} top(s_{BQ4}) & \text{if } s_{B4} = 0 \\ occ_{B4} & otherwise \end{cases}$$

$$s'_{BY} = \begin{cases} 0 & \text{if } s_{B4} = 0 \text{ AND } l_{V(top(s_{BQ4}))} = 1 \\ s_{BY} & otherwise \end{cases}$$

$$\sigma'_{BY} = \begin{cases} 0 & \text{if } s_{B4} = 0 \text{ AND } l_{V(top(s_{BQ4}))} = 1 \\ s_{BY} & otherwise \end{cases}$$

$$occ'_{BY} = \begin{cases} 0 & \text{if } s_{B4} = 0 \text{ AND } l_{V(top(s_{BQ4}))} = 1 \\ occ_{BY} & otherwise \end{cases}$$

$$s'_{BQO} = \begin{cases} s_{BQO} & \text{if } s_{B4} = 0 \\ s_{BQO}\, occ_{B4} & otherwise \end{cases}$$

$$s'_{BQ4} = \begin{cases} rest(s_{BQ4}) & \text{if } s_{B4} = 0 \\ s_{BQ4} & otherwise \end{cases}$$

. . . . . .

Figure 3.79 Local Transition for BLOCK·4

CONDITION ROUTINE FOR *BLOCK·O*:

$C_{BO}(s_{BQO}, s_{BO}) \equiv$

1. $s_{BQO} \neq \Lambda$ *AND* $s_{BO} = 0$), OR
2. $s_{BO} = 1$

ACTIVITY ROUTINE FOR *BLOCK·O*:

$$f_{BO}(s_{BO}, \sigma_{BO}, occ_{BO}, s_{B4}, \sigma_{B4}, occ_{B4}, s_{B8}, \sigma_{B8}, occ_{B8}, s_{EX}, \sigma_{EX}, s_{WT1}, s_{BQO}, \ldots) =$$
$$(s'_{BO}, \sigma'_{BO}, occ'_{BO}, s'_{B4}, \sigma'_{B4}, occ'_{B4}, s'_{B8}, \sigma'_{B8}, occ'_{B8}, s'_{EX}, \sigma'_{EX}, s'_{WT1}, s'_{BQO}, \ldots)$$

$$s'_{BO} = \begin{cases} 1 & \text{if } s_{BO} = 0 \\ 0 & \textit{otherwise} \end{cases}$$

$$\sigma'_{BO} = \begin{cases} SVC\cdot TIME(O, l_{V(top(s_{BQO}))}, d_{V(top(s_{BQO}))}) & \text{if } s_{BO} = 0 \\ 0 & \textit{otherwise} \end{cases}$$

$$occ'_{BO} = \begin{cases} top(s_{BQO}) & \text{if } s_{BO} = 0 \\ 0 & \textit{otherwise} \end{cases}$$

$$s'_{B4,B8} = \begin{cases} 0 & \text{if } s_{BO} = 0 \textit{ AND } l_{V(top(s_{BQO}))} = 1 \\ s_{B4,B8} & \textit{otherwise} \end{cases}$$

$$\sigma'_{B4,B8} = \begin{cases} 0 & \text{if } s_{BO} = 0 \textit{ AND } l_{V(top(s_{BQO}))} = 1 \\ \sigma_{B4,B8} & \textit{otherwise} \end{cases}$$

$$occ'_{B4,B8} = \begin{cases} 0 & \text{if } s_{BO} = 0 \textit{ AND } l_{V(top(s_{BQO}))} = 1 \\ occ_{B4,B8} & \textit{otherwise} \end{cases}$$

$$s'_{EX} = \begin{cases} 0 & \text{if } s_{BO} = 0 \\ 1 & \textit{otherwise} \end{cases}$$

$$\sigma'_{EX} = \begin{cases} \infty & \text{if } s_{BO} = 0 \\ 0 & \textit{otherwise} \end{cases}$$

$$s'_{BQO} = \begin{cases} rest(s_{BQO}) & \text{if } s_{BO} = 0 \\ s_{BQO} & \textit{otherwise} \end{cases}$$

$$s'_{WT1} = \begin{cases} s_{WT1} & \text{if } s_{BO} = 0 \\ s_{WT1} + (CLOCK - A_{V(occ_{BO})}) & \textit{otherwise} \end{cases}$$

. . . . . .

Figure 3.80 Local Transition for BLOCK·*O*

STATE VARIABLES:

For each *TURNERn*, the following are *state* variables:

STATE·OF·TURNERn (s_{Tn})

Tn·TIME·LEFT·IN·STATE (σ_{Tn})

CONDITION ROUTINE FOR *T*1:

$C_{T1}(s_{B4}, s_{B1}, s_{BL}, s_{BB}, s_{T1}, s_{WQ1}) \equiv$ where activating conditions are

1. $(\ s_{WQ1} \neq \Lambda\ AND\ s_{T1} = 0\ AND$
$(\ s_{B4} = 0\ AND\ s_{B1} = 0\ AND\ s_{BL} = 0\ AND\ s_{BB} = 0\ ))\ OR$
2. $s_{T1} = 1$

ACTIVITY ROUTINE FOR *T*1:

$$f_{T1}\ (\ s_{T1}, \sigma_{T1}, s_{WQ1}, s_{BQ4}\ ) = (\ s'_{T1}, \sigma'_{T1}, s'_{WQ1}, s'_{BQ4}\ )$$

$$s'_{T1} = (\ s_{T1} + 1\ )\ mod\ 2$$

$$\sigma'_{T1} = 0$$

$$s'_{WQ1} = \begin{cases} s_{WQ1} & \text{if } s_{T1} = 0 \\ rest(s_{WQ1}) & otherwise \end{cases}$$

$$s'_{BQ4} = \begin{cases} s_{BQ4} & \text{if } s_{T1} = 0 \\ s_{BQ4}\ top(s_{WQ1}) & otherwise \end{cases}$$

Figure 3.81 Local Transition for TURNER*n*

STATE VARIABLES:

$STATE{\cdot}OF{\cdot}SPLITTER(\ s_{SP}\ )$

$SP{\cdot}TIME{\cdot}LEFT{\cdot}IN{\cdot}STATE(\ \sigma_{SP}\ )$

CONDITION ROUTINE FOR *SPLITTER*:

$C_{SP}(\ s_{LQJNT}\ ,\ s_{LQ1}\ ,\ s_{LQ2}\ ,\ s_{SP}\ ) \equiv$

1. $(s_{LQJNT} \neq \Lambda\ AND\ s_{SP} = 0\ AND$
   $((\ num(\ s_{LQ1}\ ) \leq 4\ AND\ l_{V(\ top(\ s_{LQJNT}\ ))} = 1)$ OR
   $(\ num(\ s_{LQ2}\ ) \leq 4\ AND\ l_{V(\ top(\ s_{LQJNT}\ ))} = 2))),$ OR
2. $s_{SP} = 1$

ACTIVITY ROUTINE FOR *SPLITTER*:

$f_{SP}(\ s_{SP}\ ,\ \sigma_{SP}\ ,\ s_{LQJNT}\ ,\ s_{LQ1}\ ,\ s_{LQ2}\ ) \equiv$
$(\ s'_{SP}\ ,\ \sigma'_{SP}\ ,\ s'_{LQJNT}\ ,\ s'_{LQ1}\ ,\ s'_{LQ2}\ )$

$$s'_{SP} = (s_{SP} + 1)\ \ mod\ 2$$

$$\sigma'_{SP} = 0$$

$$s'_{LQJNT} = \begin{cases} s_{LQJNT} & \text{if } s_{SP} = 0 \\ rest\ (\ s_{LQJNT}\ ) & otherwise \end{cases}$$

$$s'_{LQ1} = \begin{cases} s_{LQ1}\ top\ (\ s_{LQJNT}\ ) & \text{if } s_{SP} = 1 \text{ and } l_{V(\ top(\ s_{LQJNT}\ ))} = 1 \\ s_{LQ1} & otherwise \end{cases}$$

$$s'_{LQ2} = \begin{cases} s_{LQ2}\ top\ (\ s_{LQJNT}\ ) & \text{if } s_{SP} = 1 \text{ and } l_{V(\ top(\ s_{LQJNT}\ ))} = 2 \\ s_{LQ2} & otherwise \end{cases}$$

Figure 3.82 Local Transition for SPLITTER

STATE VARIABLES:

$STATE \cdot OF \cdot EXIT(s_{EX})$

$EXIT \cdot TIME \cdot LEFT \cdot IN \cdot STATE(\sigma_{EX})$

$STATE \cdot OF \cdot TERM(s_{TERM})$

$TERM \cdot TIME \cdot LEFT \cdot IN \cdot STATE(\sigma_{TERM})$

CONDITION ROUTINE FOR *EXIT*:

$C_{EX}(s_{EX}, s_{MOD}) \equiv$

1. $(s_{EX} = 1)$ *AND* $s_{MOD} \neq 3$

ACTIVITY ROUTINE FOR *EXIT*:

$f_{EX}(s_{EX}, \sigma_{EX}, s_{LOTP}, s_{NDISS}, s_{MOD}) =$
$(s'_{EX}, \sigma'_{EX}, s'_{LOTP}, s'_{NDISS}, s'_{MOD})$

$$s'_{EX} = 0$$

$$\sigma'_{EX} = \infty$$

$$s'_{LOTP} = \begin{cases} s_{LOTP} - 1 & \text{if } s_{MOD} \leq 1 \\ s_{LOTP} & \textit{otherwise} \end{cases}$$

$$s'_{NDISS} = \begin{cases} s_{NDISS} + 1 & \text{if } s_{MOD} = 2 \\ s_{NDISS} & \textit{otherwise} \end{cases}$$

$$s'_{MOD} = \begin{cases} 0 & \text{if } s_{LOTP} > 1 \\ 1 & \text{if } s_{LOTP} = 0 \\ 2 & \text{if } s_{LOTP} = 0 \\ 3 & \text{if } s_{NDISS} > LOSS \end{cases}$$

$$s'_{WT_o} = \begin{cases} 0 \ (\textit{for all } o) & \text{if } s_{MOD} = 1 \\ s_{WT_o} & \textit{otherwise} \end{cases}$$

CONDITION ROUTINE FOR *TERM*:

$C_{TERM}(s_{MOD}) \equiv$

1. $(s_{MOD} = 3)$

ACTIVITY ROUTINE FOR *TERM*:

$f_{TERM}(s_{HALT}) = (s'_{HALT})$

$$s'_{HALT} = TRUE$$

Figure 3.83 Local Transitions for EXIT and TERM

CHAPTER 4

A COMPARATIVE REVIEW

The literature review and CF applications, discussed in Chapters 2 and 3, indicate that two basic types of guidance are provided by the commonly used CFs in constructing model representations.

First is *implementation* guidance (algorithmic, managerial, supervisory) which directly impacts the subsequent executable form of any model representation. Such guidance centers on two aspects of the model representation: a model's mode of sequencing (whether in the form of **events, activities, processes**, etc.) and its method of sequencing (e.g., whether by explicit **scheduling** of events, **scanning** of conditions, or by the concurrent control of component **interactions** utilizing a combination of scheduling and scanning techniques). This type of guidance is fundamental to achieving the translation of the representation into executable code.

Secondly, CFs can provide *design* (structural, existential, skeletal) guidance. Here, the modeler is aided in his definition of the model's static structure as he identifies the objects (components, entities) and their attributes which comprise the model. Within design guidance, the modeler is further assisted in the expression of the dynamic relationships and the rules of interaction that must exist among model objects during the progression of the model through time and state (i.e., the model's dynamic structure). The common provision of enabling the representation of relationships (or sets) which exist among model objects may be provided.

Since the boundaries among CFs are not well-defined, it is difficult to objectively

compare them. However, we strive to consider the CFs in their purest form and intention, with regard to their explicit (rather than implicit) features. We caution the reader from attempting to generalize the comparisons to all problem domains. Instead, comparative comments are made on the basis of our experience in the problem domain (Traffic Intersection) in which we performed the CF applications. In addition, since the ES, PI, and TF applications were performed under the influence of an SPL (Simulation Programming Language), we restrict our comparison to the features of the CFs which are independent of their surrounding language implementation. Therefore, in this chapter we explore the comparisons of the CFs with regards to the types of guidance that each explicitly makes available to the modeler. Grouping our comparisons by guidance type improves the clarity and meaningfulness of our review. Consequently we draw comparisons among those CFs (and only among those) which display implementation guidance, etc. The groupings of CFs by the type of guidance permitting comparisons are shown later in Figure 5.1. By means of this comparative discussion, the roles of the CFs are clarified with respect to their guidance. Thus, the basis for the development of a taxonomy in Chapter 5 is realized.

4.1 Implementation Comparisons

A discussion of the comparison among CFs that deliver implementation guidance includes an analysis of the characteristics of the ES, AS, TPA, PI, and TF CFs. These CFs, because of their very nature, form the basis for the boundaries of this comparison. We first consider the relative merits of ES, AS, and PI and then extend the comparison to include TPA and TF. Comments pertinent to other CFs are also offered.

From the introduction to this chapter, we realize that implementation guidance

influences the *mode of sequencing* of the model representation and the algorithmic strategy or *method of sequencing* that is to be employed. These constituents of implementation guidance, due to their low-level nature, have a definite impact upon the modeler. Both take part in determining efficiency in terms of building the model representation, the subsequent programming task, and its later computer execution. The mode of sequencing reflects the world view or *Weltansicht* [Lackner 1962] that is promoted by the CF in use by the modeler and the view that will be taken to effect model transformation from state to state. Viewing the model as being composed of events, activities, or processes (characteristic of the ES, AS, and PI CFs respectively) influences the programmer's task and determines the coding format and structure (e.g., whether in the form of event routines, testheads and activity routines, or process descriptions) of the programmed model. The method of sequencing found in the implementation guidance determines the data structures and list processing techniques (if any) that are necessary.

4.1.1 Aspects Concerning the Sequencing Mode

Since the ES CF dictates that the modeler use events as the principal unit for component interaction, a programmed model based on the ES CF is distinguished by event routines. The **turn.ns.green**, **arrival.lane1**, **departure**, **enter**, and the **arrival.blockd** event routines from the ES CF application in Chapter 3 are examples. The burden is placed upon the modeler to include all conditional testing (based upon conditions other than time) within these routines [Fishman 1973; Hooper 1986b]. The modeler must explicitly state, through means of this conditional testing, all consequences that will follow the occurrence of a particular event, as contained within its event routine [Neelamkavil 1987]. The events which change the light colors in the ES CF application of

Chapter 3 (Section 3.2) also include various conditional tests for controlling vehicle entry to the intersection. Figure 4.1 (taken from the **turn.ns.green** event routine) shows that after a north-south light change to green the modeler tests the entry conditions for vehicles to enter from lanes 1,2,5,6,7,8, and 11 with the "call test.entry" statements. Since the consequences of this light change to green may influence a right turn on red for lanes 5 and 11 or direct movement into the intersection from lanes 1,2,6,7, or 8, the modeler is forced to consider and check all possibilities. Also notice a similar problem for the modeler in Figure 4.2 taken from the **arrival.blockd** event routine. For vehicles that have come from lane 9, the modeler must release block N and then check several entry conditions. The simple consequence of releasing block N may satisfy the entry conditions for vehicles from any of several lanes. The burden to recognize which conditions to test lies squarely on the modeler's shoulders. Experts [Pidd 1984; Birtwistle et al. 1985; Kreutzer 1986] agree that as model complexity increases, it becomes increasingly difficult for the modeler to accurately handle these determinations and maintain consistency.

With an increase in complexity, the programmed model tends to be error-prone. Also, modifications and enhancements to the code are not easily made and debugging can be a frustrating task. Furthermore, the scheduling commands of future events are scattered throughout the code resulting in a programmed model that has fragmented logic and is difficult to read and understand [Kreutzer 1986]. All of these problems characterize the development of the ES CF application. The lack of key conditional tests early in development causes the traffic intersection to become clogged during testing. Debugging to locate such problems is extremely difficult and tedious. Once a problem is solved and corrected, it is not uncommon to find that the same correction is required in multiple locations within the code due to the fragmentation of the logic.

```
event turn.ns.green
1       let ns.color(1) = green              ''Set color attributes
2       let west.color(1) = red
3       let east.color(1) = red
4       let clearedwe = false                ''Set clearance flag to False
5       call test.entry.9.to.11(11,lane11,block.w)
6       call test.entry.345(5,lane5,block.e)
7       call test.entry.12(1,lane1,block.i) ''Test various entries
8       call test.entry.12(2,lane2,block.j)
9       call test.entry.678(6,lane6,block.c)
10      call test.entry.678(7,lane7,block.b)
11      call test.entry.678(8,lane8,block.a)
```

Figure 4.1 A Portion of Event TURN.NS.GREEN (ES CF)

```
15      case 9                                  ''Lane 9 car
16         call releese(block.n)                '' free block n, test.left
                                                '' for lane9, test.entry, and
                                                '' sched a departure
17         call test.left(from.9,lane9,block.3)
18         call test.entry.12(1,lane1,block.i)
19         call test.entry.12(2,lane2,block.j)
20         call test.entry.678(6,lane6,block.c)
21         call test.entry.678(7,lane7,block.b)
22         call test.entry.678(8,lane8,block.a)
23         schedule a departure given a.car in 0.866 seconds
```

Figure 4.2 A Portion of Event ARRIVAL.BLOCKD (ES CF)

On the other hand, when the number of objects that compose a model is manageable and the interactions among them are few, the modeler enjoys precise control over the model's execution [Kiviat 1969]. Because of the complexity of the TI, this advantage is not realized during the ES CF application.

The AS CF with its activity-orientation promotes a dramatic improvement in the modeler's ease during the programming task. The duo of testheads and activity routines frees the modeler from having to explicitly specify the interactions and relations among events [Kreutzer 1986; Laski 1965]. Therefore, the removal of this modeler responsibility for complex applications produces substantial gains in programming efficiency. The result is a model representation that is "readable, easy to design, modify, and extend" [Kreutzer 1986]. Easier top-down design with a uniform style can be achieved [Pidd 1984; Birtwistle et al. 1985]. The readability and simplification of the model derived from the AS CF primarily result from the clarity achieved through the grouping of the conditional tests [Kiviat 1969; Kreutzer 1986]. The benefits of such an approach are evident from inspection of the AS CF application pseudo-code in Chapter 3 (Section 3.3). The testheads for each of the activity routines are clear and relatively easily stated. The modeler is not entangled with the details of the consequences of state changes. This is in stark contrast to the ES CF application where such details are a major encumbrance to the modeler.

The process descriptions of the PI CF introduce a totally different approach. Shannon [1975] describes this approach as permitting conceptual "articulation". Others consider that the PI CF allows a model representation that is more natural, intuitive, understandable, and conceptually simpler [Pidd 1984; Fishman 1973; Neelamkavil 1987; Hooper and Reilly 1982; Birtwistle et al. 1985]. The modeler is able to confine all information

pertinent to a single process within its description, including the time flow data [Neelamkavil 1987].

Most often, the PI CF is used in conjunction with the OOP in performing this encapsulation of information into object modules [Kreutzer 1986]. Each module characterizes the process dynamics of a single process class. Therefore, modularization becomes an achievable feature; the model is not as fragmented as in comparable ES and AS models [Birtwistle et al. 1985]. Model logic is concentrated in a single location resulting in improved readability and understanding of model logic flow, reduced complexity, and shorter model descriptions [Kiviat 1969; Banks and Carson 1985; Kreutzer 1986]. Modularization naturally enhances maintainability and helps reduce problems in debugging the programmed model. Fishman [1973] also adds that statistics collection statements are easier to implement since they can be localized in modules rather than spread out as in an ES CF model.

The construction of the PI CF application in Chapter 3 (Section 3.5) appears to progress faster than the ES CF. The task of converting the conceptual notions of the model into the process descriptions is accomplished in a smooth fashion. The findings in the literature that characterize such an approach as "natural", "intuitive", "understandable", etc. are confirmed. Issues of modularity come to bear heavily in speeding the development of the application. Each process description clearly shows all the interactions of the vehicle and its points of conditional and unconditional delay. Corrections to the code are easily made during the accomplishment of the application. The descriptions, in some respects self-documenting, enable the modeler to maintain an effective grasp on and understanding of the finer details of the model. Even with the lapse of several months since the completion of the PI CF application, maintaining excellent comprehension on

returning to the code is experienced.

Overstreet and Nance [1986] provide an interesting conceptual summary of the ES, AS, and PI CFs on the basis of locality, or that property which is distinguished by the grouping of common information into one location. The ES CF can be considered to display locality of time in that event routines contain the related actions that are to take place in one instant. The events list then groups all simultaneous events into one location. The AS CF provides locality of state. The testheads offer a grouping of model state conditions under which associated actions are to occur. Finally, locality of object is evident in models based on the PI CF. Each process description describes the life-cycle actions of a particular class of model object.

Each perspective of locality offers a unique advantage. Clearly, the modeler must choose that locality which most benefits his cause and promotes the attainment of model objectives.

4.1.2 Aspects Concerning Sequencing Method

The sequencing methods of the ES, AS, and PI CFs primarily influence the execution efficiency of models which are constructed in accordance with their implementation guidance. This aspect of implementation guidance is hidden in most respects to the modeler when an SPL is used: the SPL largely assumes responsibility for the algorithmic strategy. Such is the case for our applications of the ES, PI, and TF CFs, which are accomplished using SIMSCRIPT, SIMULA, and GPSS/H respectively. The modeler often needs to understand the SPL's use of its sequencing method, even though it is hidden to him. The effects of this "hidden" component remain significant. Comments concerning execution efficiency in this section follow the majority view that is found in the literature and are

not based on studies performed as a result of this research. We choose not to perform such studies since applications of the CFs to the TI require extensive programming in the absence of an SPL. SPLs are not readily available for the AS and TPA CFs; their algorithmic strategies are not implemented in our applications.

As noted earlier, the sequencing guidance of the ES CF stipulates that an events list holds (ordered by time) a succession of unconditional events [Hooper 1986b]. This exact determination of event routine execution produces an efficient execution when the model is composed of less interactive, more independent components [Kiviat 1969; Shannon 1975; Birtwistle et al. 1985; Hooper 1986b]. Unlike within the AS CF, "repeated scanning is not required to determine when they [the independent events] can be done" [Kiviat 1969]. The algorithm is streamlined because the modeler assumes the burden of representing the component interactions as discussed earlier. From experience with the ES CF application, once appropriate conditional tests are inserted and consistency among them is achieved (albeit after many labor-intensive hours of work), the modeler is able to concentrate on other aspects of the model. He is able to completely divorce himself from concerns of the sequencing method. Although selection of the next event is easily done, the algorithm still handles the filing, searching, selection, creation, and destruction of event records in the events list [Fishman 1973].

Under these same circumstances (i.e., independent components), the AS CF produces an inefficient representation for execution. As you recall, all testheads are scanned during a single scanning phase. Obviously, as the number of independent components increase, the number of repetitive, redundant, and unnecessary scans also increases [Laski 1965; Kreutzer 1986; Pidd 1984; O'Keefe 1986b]. According to Kreutzer [1986], this is the "price to be paid for the convenience of declarative (conditional) scanning". However, when the

model is characterized by a large number of primarily dependent and interactive components, the AS CF demonstrates improved execution efficiency [Kiviat 1969; Hooper 1986b]. A corresponding ES CF model which has highly interactive components could spend a great deal of execution time in list processing chores. Since there is no such analogous "list", the AS CF escapes this type of work [Neelamkavil 1987] and is more attractive than the ES CF under these circumstances. The scanning and logical checks become "less time consuming" than the overhead requirements of record management and list processing [Fishman 1973]. The AS CF algorithm, unlike the ES CF, is considered to provide much more useful work for the modeler since it handles the component interactions with the scan and removes this responsibility from the modeler [Kreutzer 1986; Pidd 1984; Birtwistle et al. 1985]. Indeed, this becomes quite clear when an application, like that given in Chapter 3 (Section 3.3) is undertaken under the AS CF.

Much like the AS CF, the PI CF is able to simplify the specification of a model since the use of reactivation points enables a conditional wait capability. The interactions among components remain implicit and the modeler is eased of the burden of explicitly representing these dependencies [Blunden and Krasnow 1967]. However, the PI CF demands a much more complicated implementation due to its concurrency requirements and need to combine activity scanning and event scheduling techniques [Kreutzer 1986; Pidd 1984]. This problem can be overcome, however, when an SPL is used that implements the PI CF. It follows that we do not experience this drawback during our application of the PI CF. However, this would be readily apparent in the performance of a comparable model using some high-level language, (e.g., standard Pascal) which in its standard form has no capability for concurrency.

The PI CF is preferred when the model is composed of a "balance" of independent and dependent components [Hooper 1986b]. However, the PI CF becomes less efficient when competition for resources is dominant [O'Keefe 1986b]. We see in the next section that the competition for resources also places added requirements on the modeler.

4.1.3 Extending the Comparisons

Laski [1965] recognizes that during the update of the clock for the AS CF that the information is lost which would link the new time with associated Bound-activities. The TPA CF does not lose this information. Instead, the pending Bound-activities are clearly identified with the updated clock time. Their execution during the B-phase reduces the length of the C-phase and eliminates unnecessary scans for Bound-activities that are not pending. The application of the TPA CF in Chapter 3 (Section 3.4) demonstrates these points. The list of B-activities due for execution is derived by examining the t-cells of the B-activities shown in Section 3.4.2. The AS CF requires 89 testheads to be scanned. Elimination of the testheads of the B-activities (while using the TPA CF) reduces the number of scanned testheads to 40, a fifty-five percent reduction.

By separating the "things that must happen" (the due B-activities) from the "things that might happen" if the conditions are right (the C-activities) [Crookes et al. 1986], the TPA CF removes most of the inefficiencies of the AS CF [Tocher 1979; O'Keefe 1986b]. This proves helpful in making the model "easier to analyze, comprehend, and extend", especially when there is complex interaction and competition for resources [Crookes et al. 1986]. The TPA CF thereby retains the advantages of the AS CF while improving upon a model's execution efficiency. The incorporation of a next-event set approach (in the B-phase) and the retention of a reduced length C-phase enables the TPA CF to be efficient

for both models with relatively independent and highly dependent components [O'Keefe 1986b].

The TF CF improves upon the PI CF by automatically accomplishing "many of the tasks which fall upon the programmer" who uses a PI-based SPL like SIMULA [Birtwistle et al. 1985]. For instance, the PI CF application to the Traffic Intersection (Chapter 3, Section 3.5) demonstrates that the modeler is responsible for signaling an object's passivation and reactivation. Furthermore, it is necessary for the modeler to explicitly control the queueing up and competition for resources. This is illustrated in lines 25-40 of Figure 4.3, a portion of the process for cars from lane 8. If a vehicle cannot enter the intersection, the modeler explicitly provides for that vehicle to enter the lane queue, awaiting entry to the intersection. Once in the queue, the vehicle's process is passivated. Also, notice that when a vehicle is removed from the lane queue (line 30 or 36) and enters the intersection (line 38-40), the modeler is required to activate the first car (if any) remaining in the lane queue. Although the advantages to describing the movement of a single vehicle within a process description are significant, these details of an object's movement are difficult to specify. Indeed, the difficulties described here rival (to a lesser extent) those discussed concerning the conditional testing requirements placed on the modeler when under the ES CF. Such problems also contribute to the added length in application code over that for the ES CF.

The TF CF performs these tasks within the block structures that are provided (e.g., the block statements of GPSS). Figure 4.4 (taken from the lane 8 submodel in Chapter 3, Section 3.6) demonstrates the advantage of the TF CF in accomplishing object activation, passivation, and competition for resources for the modeler. The block statements (specifically **SEIZE** and **RELEASE**), as discussed in Section 3.6.2, automatically do

```
20    activate new car8 delay (weibl(56.0592, 0.63923, seed8));
                                                !Generate next arrival    ;
21    mydriver :- new nsdriver(tfclight, square_a, pforkandtcreek,
22          this car8);                         !Create driver            ;
23    lane := 8;                                !Set attributes           ;
24    right := true;
25    if not lane8.empty then begin             !Enter lane8 queue when   ;
26       into(lane8);                           ! cars are already in lane;
27       passivate;                             !Wait in line for turn    ;
28       activate mydriver after current;       !At head of line, turn on ;
29       passivate;                             ! driver.                 ;
30       out;
31       end
32    else if (((tfclight.south.red) or (square_a.busy) or
        ((not pforkandtcreek.nsclear) and
        (not pforkandtcreek.clearedns))) and
        ((not right) or (tfclight.south.green) or (square_a.busy) or
        (not pforkandtcreek.r8clear))) then begin
33       into(lane8);                           !Can't immediately enter  ;
34       activate mydriver after current;       ! so first in queue,      ;
35       passivate;                             ! and turn on driver.     ;
36       out;
37       end;
38    if not lane8.empty then                   !Ready to enter, so turn  ;
39       activate lane8.first after current;    ! on any car waiting in   ;
40    entered := true;                          ! lane8 queue.            ;
41    transitfm8(pforkandtcreek);               !Enter and transit        ;
```

Figure 4.3 Excerpts from the CAR8 Process (PI CF)

```
         GENERATE   1000*&WEIBL(56.0592,0.63923,MX$SEED(&I,10))
*                                 A vehicle arrives in Lane 8
         QUEUE      STAT8R        Collect statistics for 8R vehicles
         SEIZE      FRONT8        Capture front end of Lane 8
         TEST E     BV$ENTER8R,1  Wait until the vehicle can enter the
*                                 intersection from Lane 8 to turn right
         TEST E     LS$LYTENSN,1,SKIP8R      If LYTENSN is red, skip
*                                            the next LOGIC Block
         LOGIC S    CLEARNSN      Intersection clearance was checked for
*                                 NS & SN traffic when light LYTENSN
*                                 just turned green
SKIP8R   SEIZE      BLOKA         Capture block A
         RELEASE    FRONT8        Free front end of Lane 8
         ADVANCE    2153          Travel on block A
         SEIZE      BLOKK         Capture block K
         RELEASE    BLOKA         Free block A
         ADVANCE    1507          Travel on block K
         RELEASE    BLOKK         Free block K
         DEPART     STAT8R        Record collected statistics
         TERMINATE  1             Exit the intersection
```

Figure 4.4 Excerpts from the LANE8 Submodel (TF CF)

these operations. This feature significantly improves the speed in accomplishing the TF CF application over that achieved for the PI CF. However, ease in developing the process descriptions is comparable. Perhaps the singlemost distinguishing feature of the TF CF application is the tremendous reduction in the amount of code compared with the ES and PI CFs.

A block structure or chart of transaction flow is also a very natural way to represent many systems. The textual form of the block structure provides a clear and straightforward means of documenting the model [Gordon 1979]. Debugging is simplified in that errors can be isolated to a particular block. With the "block isolating" error-reporting features of GPSS/H, the time spent in debugging is reduced for the TF CF application. In some cases, the modeler is limited, however, to using the defined block structures and flexibility is lost. This limitation may also inhibit the modeler's ability to specify details of communication among components. No such problems are experienced during the TF CF application to the TI.

4.1.4 Summarizing Comparisons Based on Implementation Guidance

Tables 4.1 and 4.2 review the comparative features just discussed. Table 4.1 covers the eminent features of the CFs relative to implementation guidance. Table 4.2 gives a panorama of the key, related characteristics for complex models (i.e, models like that of the TI with many components and component interactions).

The ES, AS, TPA, PI, and TF CFs provide a wide range of implementation guidance characteristics, compared and discussed above. Although other CFs under review may be used in the context of some identifiable implementation guidance like that found in the aforementioned CFs, none contains guidance that specifies the mode and method of

Table 4.1 Eminent Features of CFs Based on Implementation Guidance

CONCEPTUAL FRAMEWORK	MODE OF SEQUENCING	METHOD OF SEQUENCING	LOCALITY
ES	event	Explicit time scheduling of events; update to next-event time on events list	time
AS	activity	Conditional scanning of state conditions; update to minimum t-cell time	state
TPA	event, activity	Explicit time scheduling of B-activities; conditional scanning of state conditions for C-activities; update to minimum t-cell time	time, state
PI	process	Explicit time scheduling of object move-times on FOL with transfer to COL and conditional scan of objects on COL; update to next object move-time on FOL	object
TF	process	Explicit time scheduling of object move-times on FOL with transfer to COL and conditional scan of objects on COL; update to next object move-time on FOL	object (transaction)

Table 4.2 Characteristics of Complex Models

CONCEPTUAL FRAMEWORK	ES	AS	TPA	PI	TF
CONDITIONS FOR MAXIMUM EFFICIENCY	Independent Objects	Dependent Objects	Independent or Dependent Objects with resource competition	Balance of Independent and Dependent Objects with low resource competition	Balance of Independent and Dependent Objects with low resource competition
BURDEN ON MODELER	High	Low	Low	Moderate*	Low
BURDEN ON EXECUTIVE	Low	High	High	High	High
MODEL LOGIC DESCRIPTION	Fragmented throughout event routines	Conditional logic concentrated at testheads	Conditional logic concentrated at testheads and determined logic concentrated at B-activities	Concentrated in modules of process descriptions	Concentrated in modules of block segments
MAINTAINABILITY	Low	High†	High†	High§	High§
NATURAL REPRESENTATION CAPABILITY	Minimal	Good	Good	Excellent	Excellent
DEVELOPMENTAL TIME, EFFORT REQUIRED	Very high	Low	Low	High	Low
APPLICATION LINES OF CODE‡	1312	-	-	1778	443

† Due to localization of state with grouping of conditional testing
§ Due to localization of object and modularization of process descriptions
* Due to modeler responsibilities in activation, passivation, and queueing for resources
‡ Lines of code for event routine or process descriptions; applicable to Chapter 3 applications only; does not include code for initialization or statistics collection

sequencing that is characteristic of pure implementation guidance. For example, one point has been made regarding the OOP in its association with the PI CF: the OOP enhances modularization of the process descriptions. In addition, a common feature of the OOP is the ability to save and restore an object's state, similar to the use of reactivation points in the PI CF. Indeed, some [Kreutzer 1986] feel that the OOP is an extension of the PI CF. However, the OOP does not explicitly possess the mode and method of sequencing in its guidance. Therefore, we do not consider it to contain implementation guidance.

While the impact of implementation guidance primarily centers on the program design and execution efficiency, the remaining sections of the comparative review deal with the conceptual and communicative [Balci 1986] design issues of model representation.

4.2 Design Comparisons

This section deals with comparing the CFs relative to their ability to effectively assist the modeler in his design of the static and dynamic structure of the model. More specifically, we seek to investigate how well each CF aids in the designation of model objects and their associated attributes and in the specification of the dynamic rules of interaction. Additionally, the identification of relationships (how the objects are "bound" or related to one another) and the description of the input/output exchange of the model with its environment are also of concern. The bonding and interfacing requirements may be of a static or dynamic nature.

4.2.1 Object and Attribute Identification

With the exception of the ES, AS, TPA, PI and TF CFs, all the CFs under review provide limited guidance for the identification of objects and their attributes. The modeler must use his understanding of the system being modeled to identify the objects and their attributes. The CFs discussed below coerce the modeler to perform this task.

- OOP — The OOP is clearly based upon the decomposition of a model into its component objects. Additionally, the OOP conceptually stipulates that all information for a given object is encapsulated within that object's description. This includes the provision, enhanced by inheritance mechanisms, for attribute identification which is required to describe an object. Modularity of an object and its attributes is important to the modeler due to conceptual clarity and maintainability issues. Inheritance eases attribute association for a modeler and is an important benefit of the OOP concerning object and attribute identification, eliminating redundancies which might otherwise be required. The **class BLOCK** and **class BLOCKA** declarations of the OOP application in Chapter 3 (Section 3.7) demonstrate the power of this feature. With 35 block descriptions to declare (blocks A-Z, blocks 1-9), the attributes of a generic block (the class BLOCK) are inherited by each individual block; the redeclaration of the generic attributes within each of the 35 blocks is not required.

- PGM — Object and attribute data are contained in the Variable Attribute and Queue Attribute Tables which are presented in the PGM application of Chapter 3 (Section 3.8).

- ERA and EAS — Both the ERA and EAS CFs derive their conceptual basis from

the objects (entities) and attributes that make up a given model. The squares (objects) and circles (value sets for attributes) demonstrate the ease in which objects and attributes are designated under the ERA CF with the aid of the entity-relationship diagram (Chapter 3, Section 3.9). Within the SIMSCRIPT preamble (Chapter 3, Section 3.10), the objects and attributes of the EAS CF application (within SIMSCRIPT) are evident.

• CM — The CM is an extension of the OOP and therefore provides for model object and attribute identification. The CM outline in Chapter 3 (Section 3.1) very clearly guides the modeler in a top-down definition of model objects and attributes, through the various submodels down to the base level (i.e., from the top-level Model to the Vehicle, Light, and Block submodels at the base level).

• SM — The elemental and generic structures of the SM enable a full designation of objects and their attributes. The genus graph (Chapter 3, Section 3.11) highlights the basic model objects of the SM CF application and also shows attribute information. Applying the SM CF for this level of design guidance is straightforward.

• CS — The object specification includes provision for object and attribute identification and establishes the static structure of the model of the TI for the CS application (Chapter 3, Section 3.12).

• STA — Model components and descriptive variables relate object and attribute information for the STA. The informal descriptions (Chapter 3, Section 3.13) are evidence of how the STA prompts the modeler for this data.

4.2.2 Dynamic Interactions

The relationships and rules of dynamic design guidance, once specified, provide the motive force for effecting the state changes among the model objects. Therefore, this aspect of guidance is critical to producing an accurate model representation. By means of constituent components or methodological guidance aimed directly at the specification of model dynamics, the CS, STA, and CM CFs provide explicit support, albeit limited, for accomplishing this task. We note that the dynamic design guidance provided by these CFs is independent of world view.

The transition specification of the CS guides a modeler for this purpose. CAPs (using Boolean expressions or sequencing primitives to generate time-based signals) contribute to the effectiveness of the transition specification. The transition specification, however, only provides limited guidance in format and syntax to the modeler. The modeler in using this guidance must depend upon his own knowledge and experience with the system under study to accurately develop the dynamic relationships. With the transition specification, the CS coerces the modeler to specify the model dynamics.

The STA via the DEVS formalism also provides dynamic design guidance. The "necessary equipment" to specify model dynamics is available to the modeler in the form of the time advance and transition functions The time advance feature is implicitly provided; the modeler provides information for the transition function. However, similar to the CS, the STA provides only limited guidance in format and syntax (notation). In addition, set theoretic notation and the intricate details of the DEVS formalism makes "using the equipment" a difficult task for the modeler. Yet, the STA "equipment", when properly specified, accomplishes the following:

- The time advance function enables the selection of the clock phase time and the update of the countdown variables.
- The transition functions (acting like event or activity routines, or process descriptions) are selected for execution, producing the state changes of the model. Tie-breaking rules are also accomodated for function selection.

Although the formal approach of DEVS makes it more unwieldy than the CS, the STA also coerces the modeler into the specification of model dynamics.

Bottom-up specification of the CM enables the specification of model dynamics. Bottom-up specification is not accomplished with the CM application. But as noted in Chapter 2, Barger [1986] conducted related research and supported bottom-up specification under the CM in her version of the Model Generator tool of the SMDE. Barger [1986] suggests possible ways to explicitly guide the modeler in the specification process using the information included in the CM top-down definition. The specification of model dynamics under the CM is less structured than that for the CS and STA.

It has been conjectured [Geoffrion 1987a; Patrick 1987] that SM can accommodate the dynamic relationships of discrete-event models. The SM application does not contain model dynamics. However, since the SM is not specifically oriented towards discrete-event models, dynamic design guidance is not explicit and applicability is yet to be shown.

No other CFs provide an explicit capability for dynamic design guidance except as provided by the modeler.

4.2.3 Hierarchical, Top-down Decomposition and Relationships

Balmer [1987] discusses hierarchical modeling based on "structures centered around activity modules." But within the context of this review, the hierarchical, top-down decomposition which is discussed is based on a system object viewpoint instead. A hierarchical decomposition capability supports the definition of *1:1* or *1:m* relationships. Hierarchical decompositions (from an object or entity viewpoint) are possible when the model is influenced by OOP, PGM, ERA, EAS, CM, SM, or STA CFs.

Wasserman [1984] defines a *hierarchy* as "a group of objects that exist in some partially ordered state such that a sub-group of objects that are all subservient to another object form a logical class." A hierarchical structure may demonstrate different forms of subservience or fundamental relations that are dependent "on the information that the hierarchy is attempting to capture" [Wasserman 1984]. Wasserman [1984] gives several common examples of these fundamental relations:

- **IS-A**, in which subordinate objects are instances of their parent object(s),
- **PART-OF**, where subordinate objects are components of their parent object(s), and
- **REPORTS-TO**, a useful relation for showing chain-of-command structures.

The inheritance features of the OOP and PGM described in their applications allow hierarchical decompositions. As discussed earlier, the **class BLOCK** and **class BLOCKA** declarations (Chapter 3, Section 3.7) demonstrate an IS-A hierarchy. The **class DIRECTION** and **class LIGHT** declarations show the creative use of the inheritance feature to generate a PART-OF hierarchy (i.e., the Light consists of four directions or parts: north, south, east, and west).

The use of entity and relationship sets enable the ERA to easily handle hierarchical decompositions. For example, the use of the entity-relationship diagram in Section 3.9 of Chapter 3 makes it easy to define the *1:m* relationship that exists between the intersection and its component blocks, a PART-OF hierarchy. An IS-A hierarchy is also implicit in that each of the "many" blocks takes on the attributes of the indicated value set for the BLOCK object. In a similar manner, the use of sets in the EAS CF makes such decomposition possible. In Figure 4.5 taken from the SIMSCRIPT preamble, lines 20-22 and 25 show how the EAS CF is comparable. Line 20 represents a PART-OF hierarchy; the light has component colors. Since **lane.queue** is a set in line 22, a *1:m* relationship (PART-OF) is established between a lane and the cars in its queue.

Both CM (with its OOP orientation) and SM (with its hierarchically organized structures) provide the flexibility of hierarchical decompositions and tout their top-down design capabilities. Under the CM, relational attributes and the concept of set allow a natural breakdown into hierarchies other than IS-A. The top-down definition of the intersection submodel in Chapter 3 is a PART-OF hierarchy. The modular structure of the SM makes it extremely flexible to form hierarchies according to any conceptual grouping or relation. The overview modular structure of the SM CF application in demonstrates this flexibility with its **&OBJECTS**, **&VEH_DAT**, **&LANE_DAT**, **&TRANS_AREA_DAT**, etc., conceptual modules.

The set orientation of the STA allows the establishment of object hierarchies. Experience from the STA application shows that the representation of sets of objects are easily shown. The **LANE.QUEUE**, **BLOCK.QUEUE**, and **WAIT.QUEUE** sequences (Chapter 3, Section 3.13) are examples. Hierarchical relationships can be conceptualized in the informal description portions for clarification. A stronger means for representing

```
19  permanent entities
20    every light has a ns.color, a west.color and a east.color
21    every block has a status, a laneuser, a turner and owns a block.queue
22    every lane owns a lane.queue
23    ...    ...    ...    ...    ...
24  temporary entities
25    every car has an arrtime, a laneid,an id and a to.right
       and may belong to a block.queue
       and may belong to a lane.queue
       ...    ...    ...    ...    ...
26    define arrtime as a real variable
30    ...    ...    ...    ...    ...
```

Figure 4.5 A Portion of the SIMSCRIPT Preamble with EAS CF Features

the hierarchical relationships between objects and attributes seems to be lacking. The CS application extends the CS in Chapter 3 (Section 3.12) to include sets with the **block**, **lane**, and **dir_lane** set objects in the object specification. Such extensions should allow hierarchical decompositions.

The ERA includes a clear representation of *m:n* relationships. Such a relationship exists between the lanes (1 through 11) and lane categories (Normal or Right). It has been noted from the literature that the EAS can accommodate *m:n* relationships but with difficulty. The EAS CF application of Chapter 3 does not directly show *m:n* relationships (e.g., the lane-to-lane category relationship). The need for declaring this relationship is overcome by in-line coding rather than use of explicit EAS CF features. Although we do not pursue *m:n* relationships with the CM application, the concepts of set objects and of relational attributes should make this possible. Within the SM application, the definition of the lane-to-lane category relationship is accomplished with limited success with the **VIRT_LANE** object. The definition of *m:n* relationships under the SM is not particularly straightforward due to the complicated rules of the SML.

Similar to the EAS application, we do not define *m:n* relationships under the STA. Instead, relationships like the lane-to-lane category relationship are accomodated through the definitions of the descriptive variables. For example, the use of this relationship to recover statistics data is accomplished by indexing the statistics component by the index variable, *o* (see Figure 4.6), which in turn ranges over values associated with the lane-to-lane category relationship. The set object **dir_lane** (see Figure 4.7) under the CS application is used to convey this same lane-to-lane category relationship information. Note that **dir_lane** objects are identified by indices in the range **dir_lane_range**, reflecting a similar technique to that used under the STA.

Descriptive Variables

Describing
$STATISTICS{\cdot}o$ ($o = 1,2,2R,3,4,5,5R,6,7,8,9,10,11,11R$)
$WAIT{\cdot}TIME{\cdot}o \rightarrow$ with range
R_0^+ , (s_{WTo})

Figure 4.6 A Portion of the STA CF Informal Description

```
(From Enumerated Type Description)
{ Type Name                          Definition                                  }

dir_lane_range          =      (N1, N2, R2, N3, N4, N5, R5, N6,
                                N7, N8, N9, N10, N11, R11);


(From Object Specification)
{ Object    ::          Attribute          :          Type }

dir_lane    ::       tot_wait_time         : nonnegative real;
                     deps                  : nonnegative integer;
                     exp_wait_time         : nonnegative real;


(From Initialization Transition Specification)
  FOR k := N1 TO R11 DO
      CREATE ( dir_lane [k] );
      dir_lane [k].tot_wait_time := 0;
      dir_lane [k].exp_wait_time := 0;
      dir_lane [k].deps := 0;
      END FOR
```

Figure 4.7 Excerpts from CS Application

Since an object may represent a set of objects and a process graph node may represent an underlying network of process graph nodes, we believe that the OOP and PGM also allow definition of *m:n* relationships. However, the OOP and PGM applications in Chapter 3 do not demonstrate this.

Although our experience from applications to the TI system is limited concerning abilities of the CFs for *m:n* relationships, we offer the following perceptions:

- ERA offers the most straightforward approach for *m:n* relationships when assisted by the entity-relationship diagram.
- CM and SM suggest excellent capabilities for *m:n* relationships based on the experience gained from the literature review and in performance of Chapter 3 applications. Ease in use of the SM within the SML is limited.
- EAS, STA, and CS allow definition of *m:n* relationships but without the direct, natural clarity of the above approaches.
- OOP and PGM should permit designation of *m:n* relationships.

4.2.4 Explicit Input/Output Specification

The CS and STA both contain explicit requirements for input and output specification. For the CS, the input, output, and report specification serve this requirement. The INPUT, OUTPUT, and output function components of the DEVS formalism provide this facility for the STA. Model parameters (e.g., **LOSS**, **DELAY** in the STA application of Chapter 3, Section 3.13) contribute to the input/output specification under the STA. The CM outline includes a section for interaction with the environment which also serves this function. Section II of the CM outline (see Section 3.1 in Chapter 3), enti-

tled Modeling Environment, covers model boundaries, input description, and output decisions. Other CFs, through the use of object and attribute facilities, may provide a similar result; however, the requirement is not explicit.

4.2.5 Summarizing Comparisons Based on Design Guidance

Table 4.3 outlines the comparisons of CFs based on design guidance. Each CF that has been considered in this section provides a level of design guidance that is sufficient to adequately define model structure. Depending on the aspect, certain CFs maintain a clear advantage for the modeler.

Table 4.3 Comparisons Based on Design Guidance

CONCEPTUAL FRAMEWORK	OOP	ERA	EAS	CM§	SM§	CS§	STA§	PGM§
OBJECT NAMING	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ATTRIBUTE NAMING	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
CAPABILITY FOR DYNAMIC DESIGN SPECIFICATIONS	No	No	No	Limited	No	Limited	Limited	No
TOP-DOWN HIERARCHICAL DECOMPOSITION	Yes	Yes	Yes	Yes	Yes	Yes‡	Yes	Yes
CAPABILITY FOR *MANY-MANY* RELATIONSHIPS	Yes†	Excellent	Yes†	Excellent	Good	Limited	Limited	Yes†
EXPLICIT INPUT/OUTPUT SPECIFICATION	No	No	No	Yes	No	Yes	Yes	No

† Not observed
‡ With set extension
§ Includes documenting features

CHAPTER 5

A TAXONOMY OF CFs

Based upon the comparative review of Chapter 4 we consolidate the results into a single table, Table 5.1, which places each CF according to the type of guidance that it provides. From this vantage point, we are able to step back from the details of the comparison and to grasp a broader appreciation and perspective of the CFs under review. In first considering the capabilities of the CFs with regard to the type of guidance provided, we notice varying levels of modeler support. Furthermore, we see that CFs may be categorized by the range of guidance provided. The development of a taxonomy of CFs is naturally focused on guidance types, perceived levels of modeling support, and the range of guidance.

5.1 Taxonomy Base Categories

The foundation for the categories of the taxonomy is derived from the types of guidance that a CF provides. CFs may be classified, therefore, as *implementation* or *design* CFs.

An implementation CF is defined as one providing guidance that determines the mode and method of model sequencing. The mode and method of sequencing within implementation guidance suggest that CFs may also be distinguished as:

- *event-oriented* — having the **event** as the mode of sequencing and **explicit scheduling of events** within its method of sequencing,
- *activity-oriented* — having the **activity** as the mode of sequencing and **condi-**

Table 5.1 Classifications of the CFs Under Review

IMPLEMENTATION	DESIGN (STATIC)	DESIGN (DYNAMIC)
ES	EAS	CM
AS	ERA	CS
TPA	CM	STA
PI	SM	
TF	OOP	
	PGM	
	CS	
	STA	

tional scanning of state conditions within its method of sequencing, or

• *process-oriented* — having the **process** as the mode of sequencing and **scheduling or scanning of objects** within its method of sequencing.

The design CF contains guidance that assists the modeler in defining and specifying the model static and dynamic structure. Based upon the comparative discussion in Chapter 4, it follows that design guidance also contains two sub-categories, *static* and *dynamic*.

• *static* — providing guidance which aids the definition of model static structure.

• *dynamic* — providing guidance that guides the modeler in specifying model dynamics.

Notice in Table 5.1 that the ES CF is an implementation CF while CS is both a static design CF and a dynamic design CF. Figure 5.1 shows the resulting taxonomy tree.

We noted earlier that the boundaries among CFs (based upon these categories alone) are not well defined. A taxonomy must necessarily include additional categorizations to allow further clarification where overlaps occur. These additional categorizations are now introduced to the taxonomy.

5.2 Support Level Categories

Implementation guidance, as discussed in Chapter 4, includes guidance that directly relates to the programmed execution of the model. As such, the modeling routine formats (as derived from the sequencing mode) and the model executive or monitor structure (the method of sequencing or the algorithmic strategy) represent the lowest level aspects of the

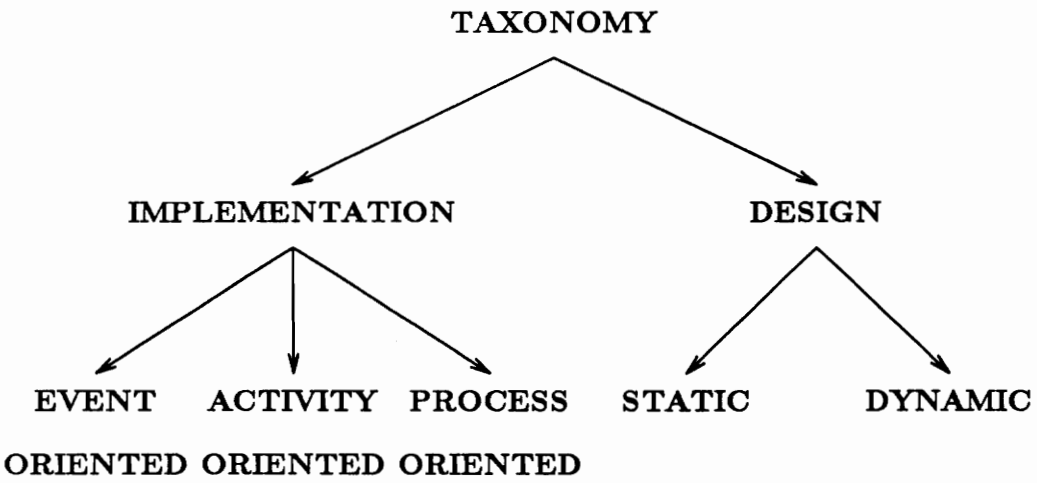


Figure 5.1 The Taxonomy Tree

model. Low-level guidance requires more intimate involvement by and retrieval of details from the modeler. Issues of syntax, etc., are also at a low-level. In general, we seek to shield the modeler from such low-level participation in order that he or she may devote full attention to the model at a higher level, free from the entanglement of details.

The CFs that provide dynamic design guidance tend to be characterized by both low-level and high-level directions. For example, CS offers high-level guidance for the specification of model dynamics as imposed by the transition specification requirements. However, the transition specification also forces the modeler to a low-level with its syntax requirements for the construction of the Condition Action Pairs, CAPs (use of sequencing primitives, etc.). A similar argument can be made concerning the STA. In this regard, the CM's flexibility helps to keep the modeler at a higher level. Dynamic design guidance, therefore, typically occurs with both low and high-level components and represents a conceptual bridge between low and high-level requirements that are placed on the modeler.

In general, the highest level of guidance for the modeler is that found within available static design guidance, applied to representing the model's static structure. At this level, the modeler is completely unencumbered with implementation details and focuses strictly on the model's static representation.

Figure 5.2 summarizes the notions of variations in support level. On the basis of this perspective, CFs may be classified as *low-level* or *high-level* CFs. When a CF contains guidance with both low and high-level components of support, such a CF is referred to as being a *mid-level* CF.

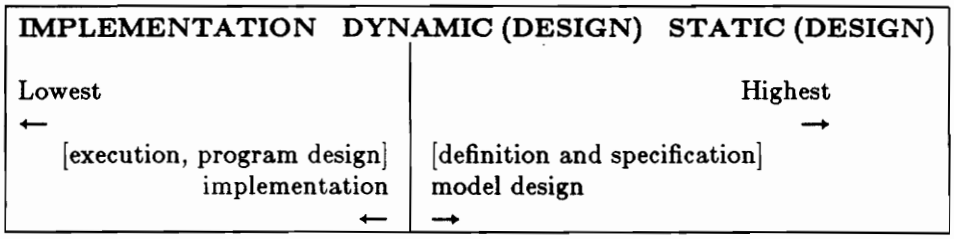


Figure 5.2 Low-level versus High-level Guidance

5.3 Range Capabilities and Resulting Categories

We speculate that a model representation must include the data derived from some form of implementation, and static design and dynamic design guidance if it is to be fully translatable into executable code. A CF which provides all three guidance types is considered to be a *full-range* CF in that it makes the "full-range" of guidance capability available to the modeler. Such a CF, if it were to exist, would provide significant advantages to the modeler. There are, however, no known full-range CFs.

The capabilities of a full-range CF are instead provided by *composite* CFs. Overstreet and Nance [1986] and Zeigler [1976] discuss at length how the CS and STA CFs may be adapted into implementation CFs (ES, AS, or PI). By transforming these CFs (the CS and STA, both of which are static design and dynamic design) to include implementation details, a composite CF is formed. Therefore, a composite CF is by definition one constructed from the combination of two or more CFs that provide distinct types of guidance. The use of an SPL, for example, by the modeler can be considered to be the implicit use of a composite CF. The SPL provides some type of implementation guidance (e.g., SIMULA provides the PI CF) and the data necessary for the static structural definition and the dynamic structural specification is modeler-defined through his use of the available primitives of the SPL.

The preceding discussion infers that a composite CF may be derived from the base guidance types and may not contain full-range capabilities. Because such a CF does not contain the full-range of guidance and contains only parts of the whole, it is considered to be *fragmentary*. Note that every CF is fragmentary.

5.4 Summary of Taxonomy Categories

CFs may be categorized on the basis of their guidance, i.e., as implementation (event-oriented, activity-oriented, or process-oriented), static design, or dynamic design. Level of support to the modeler determines whether a CF is classified as low, mid, or high-level. CFs may also be labeled as composite, fragmentary or full-range, depending on the range of guidance that they provide. Table 5.2 summarizes the terminology which has been developed for the taxonomy.

Table 5.2 Definitions of Categories of the CF Taxonomy

CATEGORY	DEFINITION
Implementation	Provides guidance that determines the mode and method of sequencing the model.
Event-oriented	Having the event as the mode of sequencing and explicit scheduling of events within the method of sequencing.
Activity-oriented	Having the activity as the mode of sequencing and conditional scanning of state conditions within the method of sequencing.
Process-oriented	Having the process as the mode of sequencing and scheduling or scanning of objects within the method of sequencing.
Design	Provides guidance that assists the modeler in defining and specifying model static or dynamic structure.
Static	Provides guidance which aids the definition of model static structure.
Dynamic	Provides guidance that guides the modeler in specifying the model dynamics.
Low-level	Provides low-level support to the modeler with particular emphasis on implementation details.
High-level	Provides high-level support to the modeler with particular emphasis on model design.
Mid-level	Provides both low and high level components of modeling support.
Full-range	Provides a minimum of implementation, static design, and dynamic design guidance.
Composite	Constructed from the combination of two or more CFs that provide distinct base types of guidance.
Fragmentary	Provides guidance support that is less than full-range in capability.

CHAPTER 6

CONCLUSIONS AND SUMMARY

This research contributes a comprehensive comparative review of CFs which is based on their individual application to a complex modeling problem, modeling the Traffic Intersection. Several represent a first-time application to this type of problem. In addition, a taxonomy of CFs is developed. The significant benefits of this research include determination of those features that are desired in a CF, improved knowledge of the types of guidance available to the modeler, insights into the information which is required from the modeler during the model design process, and implications for future research in CF development.

6.1 Characteristics of a Next-Generation CF

In Chapter 1 we noted that the CF or CFs for the SMDE MG tool must permit development of representations that will enable the subsequent development of model specifications which are analyzable, domain-independent, and fully translatable. The following features are desirable in any CF which is to accomplish these objectives for the realization of the automation-based paradigm [Balzer et al. 1983]. We discuss these features and offer comment on the current status of their availability among today's CFs.

- High-level — This feature supports the ease of use which will undoubtedly characterize CFs of the future. With a high-level CF, the use of simulation for discrete-event systems will be available to a larger audience. Certainly, we must provide CFs which can be used by modeler's who are not programmers or simulation experts. For example, the low-level features of the SM, CS, and STA CFs make

their direct use by the modeler an extremely difficult task. CFs which support high-level features are often relegated to static design guidance only.

• Independent of Domain — This feature will support domain independence requirements of resulting specifications. The modeler will be able to remain at a higher level, removed from world view considerations. Success in studying a particular problem domain is closely tied to the choice of implementation guidance for the model. The implementation guidance rather than static design or dynamic design guidance determines the world view. Currently, we see from the literature and from our experience that the CS, STA, and CM CFs are apparently free of ties to world view and can be transformed to suit a particular view, suggesting a tendency toward domain independence. However, the low-level features of the CS and STA are again highlighted with concern.

• Natural for Model Representation — Here, we consider that such a CF will produce a representation which will enable (from both static and dynamc information) the realization of a usable specification. The OOP, although limited in naturally representing other than IS-A hierarchies, brings substantial utility to the modeler through inheritance and encapsulation. Given current trends, future CFs will most likely be based upon the OOP. The OOP, although well suited to a PI- and TF-based representation, is not easily adaptable to other implementation CFs for the accomodation of different world view orientations. This issue is necessarily a problem which must be dealt with if a singular (OOP-based) CF must be relied on for general application to any problem domain.

• Broad Range of Guidance Support — In order to permit translation into executable code, the CF or CFs must guide in both the model static and dynamic design,

and in the implementation as well. This range of guidance is currently not available from a single CF. This is not necessarily a problem since composite CFs which offer static and dynamic design and implementation guidance are easily derived.

6.2 The Role of CFs

The role of CFs can be characterized as being two-fold: providing guidance to the modeler and information retrieval for the express purpose of developing a usable model specification. Both of these areas are considered strongly linked to the base guidance categories (implementation, static design, and dynamic design) of the taxonomy which has been developed. In the case of the first role, this link is obvious; the provision of guidance has been typed and classified by the taxonomy. In the latter case, as the modeler is guided in the model representation, the guidance must be sufficient to match the modeler's level of expertise and to enable the retrieval of information sufficient for the development of a model specification. Therefore, it is not surprising that the two roles work hand-in-hand with the success of the second role depending heavily upon the capability of the first. The interface (like the MG tool) between the modeler and the CF becomes critical in appropriating the capabilities of the CF and in transporting the information from the mind of the modeler to the final specification.

The applications of Chapter 3 and the comparative review of Chapter 4 leads to the following conclusions concerning the observed roles of the CFs under review.

- The implementation CFs (namely the ES, AS, TPA, PI, and TF CFs) were shown to deliver excellent guidance to the modeler.
- For best performance, the implementation guidance which is chosen by the modeler should be matched to the problem domain and level of model component

interaction. This matching could possibly be delayed until after some type of analysis of the model design.

- To keep the modeler free of the low-level details of the implementation CF, the interface and knowledge-based "participating assistant" [Balzer et al. 1983] must be heavily utilized to create the implementation level details that can be transformed into formatted code (event routines, process descriptions, etc.) and efficient algorithmic strategies.

- The issues of locality indicate that a CF must effectively retrieve information pertinent to time, state, and object localities.

- Improvements are required for static design and dynamic design guidance. Current approaches are manual-based and require heavy low-level modeler involvement. With regard to dynamic design guidance, only CS, CM, and STA CFs provide limited support in this area.

6.3 Areas of Future Research

This work suggests future areas of research aimed at the eventual development of a new CF philosophy (applicable for the SMDE MG tool), namely:

- the study of inheritance mechanisms — especially directed at improvements in representing *m:n* relationships and the various hierarchical relationships,

- investigation into the requirements for specification analysis — a review of existing analysis techniques and their distinguishing features,

- a review of the domains of applicability — determining the required range of genericity may suggest other features necessary in CFs,

- the study of the issues of the knowledge-based assistant — particularly in the areas of matching the domain to world view, transforming the the representation to a specific world view, and aiding the modeler in static design and dynamic design representation,
- the development of an integrating CF or CFs which will contain the desirable characteristics, and
- the development of the interface requirements for the new CF or CFs.

6.4 Summary

The research reported in this thesis has clarified the differences that exist among the myriad of CFs that are in use today. In particular, the comparative review highlights the significant CF features that are necessary for successful model representation of discrete-event systems. The taxonomy provides a useful and meaningful classification of CFs and produces insights into the conceptual relationships that exist among them. The characteristics of a CF or CFs that will effectively support the SMDE MG tool are identified. The roles of CFs are better understood and specific potential directions for future research are pinpointed.

BIBLIOGRAPHY

Arthur, J.D., R.E. Nance, and S.M. Henry (1986), "A Procedural Approach to Evaluating Software Development Methodologies: The Foundation," Technical Report SRC-86-008, Department of Computer Science, Virginia Tech, Blacksburg, Va., Sept.

Bagrodia, R.L, K.M. Chandy, and J. Misra (1987), "A Message-Based Approach to Discrete-Event Simulation," *IEEE Transactions on Software Engineering SE-13*, 6 (June), 654-665.

Balci, O. (1986), "Guidelines for Successful Simulation Studies: Part I and II", Technical Report TR-85-2, Department of Computer Science, Virginia Tech, Blacksburg, Va., Sept.

Balci, O. (1988), "The Implementation of Four Conceptual Frameworks for Simulation Modeling in High-Level Languages," In *Proceedings of the 1988 Winter Simulation Conference* (San Diego, Calif., Dec. 12-14). To appear.

Balci, O. and R.E. Nance (1985), "Formulated Problem Verification as an Explicit Requirement of Model Credibility," *Simulation 45*, 2(Aug.), 76-86.

Balci, O. and R.E. Nance (1987a), "Simulation Model Development Environments: A Research Prototype," *Journal of the Operational Research Society 38*, 8 (Aug.), 753-763.

Balci, O. and R.E. Nance (1987b), "Simulation Support: Prototyping the Automation-Based Paradigm," In *Proceedings of the 1987 Winter Simulation Conference* (Atlanta, Ga., Dec. 14-16). IEEE, Piscataway, N.J., pp. 495-502.

Balmer, D.W. (1987), "Software Support For Hierarchical Modelling," Technical Report, Department of Statistical and Mathematical Sciences, London School of Economics and Political Science, London, England.

Balzer, R., Cheatham, T.E., and Green, C. (1983), "Software Technology in the 1990's: Using a New Paradigm," *Computer 16*, 11 (Nov.), 39-45.

Banks, J. and J. S. Carson,II (1985), "Process-interaction Simulation Languages," *Simulation 44*, 5 (May), 225-235.

Barger, L.F. (1986), "The Model Generator: A Tool for Simulation Model Definition, Specification, and Documentation," Master's Thesis, Department of Computer Science, Virginia Tech, Blacksburg, Va., Aug.

Barger, L.F. and R.E. Nance (1986), "Simulation Model Development: System Specification Techniques," Technical Report SRC-86-005, Department of Computer Science, Virginia Tech, Blacksburg, Va.

Bauman, R. and T.A. Turano (1986), "Production Based Language Simulation of Petri Nets," *Simulation 47*, 5 (Nov.), 191-198.

Birtwistle, G., G. Lomow, B. Unger, and P. Luker (1984), "Process Style Packages for Discrete Event Modeling: Data Structures and Packages in SIMULA," *Transactions of the Society for Computer Simulation 1*, 1 (May), 61-82.

Birtwistle, G., G. Lomow, B. Unger, and P. Luker (1985), "Process Style Packages for Discrete Event Modeling: Experience from the Transaction, Activity, and Event Approaches," *Transactions of the Society for Computer Simulation 2*, 1 (May), 27-56.

Birtwistle, G.M., O.J. Dahl, B. Myhrhaug, and K. Nygaard (1979), *Simula Begin*, (2nd ed.), Van Nostrand Reinhold, New York.

Blunden, G.P. (1968), "Implicit Interaction in Process Models," In *Simulation Programming Languages: Proceedings of the IFIP Working Conference on Simulation Programming Languages* (Oslo,Norway, May, 1967). North-Holland, Amsterdam, pp. 283-287.

Blunden, G.P. and H.S. Krasnow (1967), "The Process Concept as a Basis for Simulation Modeling," *Simulation 9*, 2 (Aug.), 89-93.

Buxton, J.N. (1966), "Writing Simulations in CSL," *The Computer Journal 9*, 2 (Aug.), 137-143.

Buxton, J.N. and J.G. Laski (1962), "Control and Simulation Language," *The Computer Journal 5*, (Apr. 1962-Jan. 1963), 194-199.

Chen, P.P. (1976), "The Entity-Relationship Model- Toward a Unified View of Data," *ACM Transactions on Database Systems 1*, 1 (Mar.), 9-36.

Chen, P.P. (1983), "A Preliminary Framework for Entity-Relationship Models," In *Entity-Relationship Approach to Information Modeling and Analysis: Proceedings of the Second International Conference on Entity-Relationship Approach* (Washington, D.C., Oct. 12-14, 1981). North-Holland, Amsterdam, pp. 19-23.

Clementson, A.T. (1966), "Extended Control and Simulation Language," *The Computer Journal 9*, 3 (Nov.), 215-220.

Clementson, A.T. (1978), "Extended Control and Simulation Language," In *Proceedings of the 1978 UKSC Conference on Computer Simulation* (Chester, England, Apr. 4-6). IPC Science and Technology Press, Guildford, England, pp. 174-180.

Concepcion, A.I. and B.P. Zeigler (1988), "DEVS Formalism: A Framework for Hierarchical Model Development," *IEEE Transactions on Software Engineering 14*, 2 (Feb.), 228-241.

Cox, B.J. (1986), *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, Mass.

Crookes, J.G. (1982), "Simulation in 1981," *European Journal of Operational Research 9*, 1, 1-7.

Crookes, J.G., D.W. Balmer, S.T. Chew, and R.J. Paul (1986), "A Three-Phase Simulation System Written in Pascal," *Journal of Operational Research Society 37*, 6 (June), 603-618.

CACI, Inc. (1983), *SIMSCRIPT II.5 Reference Handbook*, J.E. Braun, Ed. CACI, Inc.-Federal, Los Angeles, Calif.

Date, C.J. (1986), *An Introduction to Database Systems (Volume I)*, Addison-Wesley, Reading, Mass.

Davies, R. and R. O'Keefe (1987), *Simulation Modelling with Pascal*, Draft manuscript, forthcoming.

DeCarvalho, R.S. and J.G. Crookes (1976), "Cellular Simulation," *Operational Research Quarterly 27*, 1, 31-40.

Dos Santos, C.S., E.J. Neuhold, and A.L. Furtado (1980), "A Data Type Approach to the Entity-Relationship Model," In *Entity-Relationship Approach to Systems Analysys and Design: Proceedings of the International Conference on Entity-Relationship Approach to Systems Analysis and Design* (Los Angeles, Calif., Dec. 10-12, 1979). North Holland, Amsterdam, pp. 103-119.

Emshoff, J.R. and R.L. Sisson (1970), *Design and Use of Computer Simulation Models*, Macmillan Publishing Co., New York.

Fishman, G.S. (1973), *Concepts and Methods in Discrete Event Digital Simulation*, John Wiley and Sons, New York.

Frankowski, E.L. and W.R. Franta (1980), "A Process Oriented Simulation Model Specification and Documentation Language," *Software -- Practice and Experience 10*, 9 (Sept.), 721-742.

Franta, W.R. (1977), *The Process View of Simulation*, North Holland Publishing, Amsterdam.

Franta, W.R. (1978), "SIMULA Language Summary," *ACM SIGPLAN Notices 13*, 8 (Aug.), 243-244.

Geoffrion, A.M. (1987a), "An Introduction to Structured Modeling," Working Paper Number 338, Western Management Science Institute, University of California, Los Angeles, Calif., Feb.

Geoffrion, A.M. (1987b), "Modeling Approaches and Systems Related to Structured Modeling," Working Paper Number 339, Western Management Science Institute, University of California, Los Angeles, Calif., Feb.

Geoffrion, A.M. (1987c), "The Theory of Structured Modeling," Working Paper Number 346, Western Management Science Institute, University of California, Los Angeles, Calif., May.

Geoffrion, A.M. (1988), "SML: A Language for Structured Modeling," Draft Working Paper, Western Management Science Institute, University of California, Los Angeles, Calif., Jan.

Golden, D.G. (1986), "Software Engineering Considerations for the Design of Simulation Languages," *Simulation 45*, 4 (Oct.), 169-178.

Gordon, G. (1975), *The Application of GPSS V to Discrete System Simulation*, Prentice-Hall, Englewood Cliffs, N. J.

Gordon, G. (1979), "The Design of the GPSS Language," In *Current Issues in Computer Simulation*, N.R. Adam and A. Dogramaci, Eds., Academic Press, New York, pp. 15-25.

Hartson, H. Rex (1987), "Introduction to Relational Database Management Systems," Course notes for CS5361 (Winter 1988), Department of Computer Science, Virginia Tech, Blacksburg, Va., Nov.

Henriksen, J.O. and R.C. Crain (1983), *General Purpose Simulation System/H (GPSS/H) User's Manual*, (2nd ed.), Wolverine Software Corporation, Annandale, Va., Feb.

Hillson, R. (1987), "Processing Graph Architectures," In *Proceedings of the 1987 Summer Computer Simulation Conference* (Montreal, Quebec, July 27-30). The Society for Computer Simulation, San Diego, Calif.

Hooper, J. W. (1986a), "Activity Scanning and the Three-Phase Approach," *Simulation 47*, 5 (Nov.), 210-211.

Hooper, J.W. (1986b), "Strategy-related Characteristics of Discrete-event Languages and Models," *Simulation 46*, 4 (Apr.), 153-159.

Hooper, J.W. and K.D. Reilly (1982), "An Algorithmic Analysis of Simulation Strategies," *International Journal of Computer and Information Sciences 11*, 2, 101-122.

Hutchinson, G.K. (1975), "Introduction to the Use of Activity Cycles as a Basis for System's Decomposition and Simulation," *ACM SIGSIM Simuletter 7*, 1 (Oct.), 15-20.

Kafura, D. (1987), "Object-Oriented Programming," Class Notes, CS5980 (Spring, 1987), Department of Computer Science, Virginia Tech, Blacksburg, Va.

Kaplan, D.J. (1987), "The Process Graph Method, An Iconic Method of Controlling Networks of Processors," In *Proceedings of the 1987 Summer Computer Simulation Conference* (Montreal, Quebec, July 27-30). The Society for Computer Simulation, San Diego, Calif.

Karp, R.M. and R.E. Miller (1966), "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal on Applied Mathematics 14*, 6(Nov.), 1390-1411.

Kelley, D.H. and J.N. Buxton (1962), "Montecode - An Interpretive Program for Monte Carlo Simulations," *The Computer Journal 5*, (Apr. 1962-Jan. 1963), 88-93.

Kiviat, P.J. (1967), "Digital Computer Simulation: Modeling Concepts," Memorandum RM-5378-PR, The Rand Corporation, Santa Monica, Calif., Aug.

Kiviat, P.J. (1969), "Digital Computer Simulation: Computer Programming Languages," Memorandum RM-5883-PR, The Rand Corporation, Santa Monica, Calif., Jan.

Kiviat, P.J., H. Markowitz, and R. Villanueva (1983), *SIMSCRIPT II.5 Programming Language* (Revised), E. C. Russell, Ed. CACI, Inc.-Federal, Los Angeles, Calif.

Kreutzer, W. (1986), *System Simulation: Programming Styles and Languages*, Addison-Wesley, Reading, Mass.

Lackner, M.R. (1962), "Toward a General Simulation Capability," In *Proceedings of the AFIPS 1962 Spring Joint Computer Conference 21* (San Francisco, Calif., May 1-3). National Press, Palo Alto, Calif., pp. 1-14.

Lackner, M.R. (1965), "A Process Oriented Scheme for Digital Simulation Modeling," In *Proceedings of the 1965 IFIP Conference 2*, (New York City, May 24-29). Spartan Books, Washington, D.C., pp.413-414.

Laski, J.G. (1965), "On Time Structure in (Monte Carlo) Simulations," *Operational Research Quarterly 16*, 3(Sept.), 329-339.

Lavender, G. (1987), "Different Inheritance Mechanisms," Presentation for CS5980, Department of Computer Science, Virginia Tech, Blacksburg, Va., May.

Malhotra, A., H.M. Markowitz, and D.P. Pazel (1982), "The EAS-E Programming Language," Research Report RC 8935 (39133), Computer Science, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., Aug.

Markowitz, H.M., A. Malhotra, and D.P. Pazel (1983), "The ER and EAS Formalisms for System Modeling and the EAS-E Language," In *Entity-Relationship Approach to Information Modeling and Analysis: Proceedings of the Second International Conference on Entity-Relationship Approach* (Washington, D.C., Oct. 12-14, 1981). North-Holland, Amsterdam, pp. 29-48.

Markowitz, H.M., A. Malhotra, and D.P. Pazel (1984), "The EAS-E Application Development System: Principles and Language Summary," *Communications of the ACM 27*, 8 (Aug.), 785-799.

Mathewson, S.C. (1974), "Simulation Program Generators," *Simulation 23*, 6 (Dec.), 181-189.

McCormack, W.M. and R.G. Sargent (1981), "Analysis of Future Event Set Algorithms for Discrete Event Simulation," *Communications of the ACM 24*, 12 (Dec.), 801-812.

McFarland, G. (1986), "The Benefits of Bottom-up Design," *ACM SIGSOFT Software Engineering Notes 11*, 5 (Oct.), 43-51.

Meyer, B. (1987), "Reusability: The Case for Object-Oriented Design," *IEEE Software*, (Mar.), 50-64.

Nance, R.E. (1971), "On Time Flow Mechanisms for Discrete System Simulation," *Management Science 18*, 1 (Sept.), 59-73.

Nance, R.E. (1977), "The Feasibility of and Methodology for Developing Federal Documentation Standards for Simulation Models," Final Report prepared for National Bureau of Standards, Department of Computer Science, Virginia Tech, Blacksburg, Va., Dec.

Nance, R.E. (1979), "Model Representation in Discrete Event Simulation: Prospects for Developing Documentation Standards," In *Current Issues in Computer Simulation*, N.R. Adam and A. Dogramaci, Eds., Academic Press, New York, pp. 83-96.

Nance, R.E. (1981a), "Model Representation in Discrete Event Simulation: The Conical Methodology," Technical Report CS81003-R, Department of Computer Science, Virginia Tech, Blacksburg, Va., Mar.

Nance, R.E. (1981b), "The Time and State Relationships in Simulation Modeling," *Communications of the ACM 24*, 4 (Apr.), 173-179.

Nance, R.E. (1986), "The Conical Methodology: A Framework for Simulation Model Development," In *Proceedings of the Conference on Methodology and Validation* (1987 Eastern Simulation Conference, Orlando, Fla., April 6-9). The Society for Computer Simulation, San Diego, Calif., pp. 38-43.

Nance, R.E. (1987), Personal Communication, Department of Computer Science, Virginia Tech, Blacksburg, Va., Sept.

Nance, R.E. (1988), Personal Communication, Department of Computer Science, Virginia Tech, Blacksburg, Va., May.

Nance, R.E. and C.M. Overstreet (1986), "Diagnostic Assistance Using Digraph Representations of Discrete Event Simulation Model Specifications," Technical Report SRC-86-001, Department of Computer Science, Virginia Tech, Blacksburg, Va., Mar.

Nance, R.E., A.L. Mezaache, and C.M. Overstreet (1981), "Simulation Model Management: Resolving the Technological Gaps," In *Proceedings of the 1981 Winter Simulation Conference* (Atlanta, Ga., Dec. 9-11). IEEE , Piscataway, N.J., pp. 173-179.

Nance, R.E., O. Balci, and R.L. Moose, Jr. (1984), "Evaluation of the UNIX Host for a Model Development Environment," In *Proceedings of the 1984 Winter Simlation Conference* (Dallas, Tex., Nov. 28-30). IEEE, Piscataway, N.J., pp. 577-584.

Neelamkavil, F. (1987), *Computer Simulation and Modelling*, John Wiley and Sons, New York.

Nygaard, K. and O. Dahl (1978), "The Development of the SIMULA Languages," *ACM SIGPLAN Notices 13*, 8 (Aug.), 245-272.

O'Keefe, R. and R. Davies (1986), "A Microcomputer System for Simulation Modelling," *European Journal of Operational Research 24*, 1, 23-29.

O'Keefe, R.M. (1986a), "Simulation and Expert Systems - A Taxonomy and Some Examples," *Simulation 46*, 1(Jan.), 10-16.

O'Keefe, R.M. (1986b), "The Three-Phase Approach: A Comment on Models'," *Simulation 47*, 5 (Nov.), 208-210.

Oldfather, P.M., A.S. Ginsberg, and H.M. Markowitz (1966), "Programming by Questionnaire: How to Construct a Program Generator," Memorandum RM-5129-PR, The Rand Corporation, Santa Monica, Calif., Nov.

Oren, T.I. and B.P. Zeigler (1979), "Concepts for Advanced Simulation Studies," *Simulation 32*, 3 (Mar.), 69-82.

Overstreet, C.M. (1982), "Model Specification and Analysis for Discrete Event Simulation," PhD Dissertation, Department of Computer Science, Virginia Tech, Blacksburg, Va., Dec.

Overstreet, C.M. and R.E. Nance (1985), "A Specification Language to Assist in Analysis of Discrete Event Simulation Models," *Communications of the ACM 28*, 2 (Feb.), 190-201.

Overstreet, C.M. and R.E. Nance (1986), "World View Based Discrete Event Model Simplification," In *Modelling and Simulation Methodology in the Artificial Intelligence Era*, M.S. Elzas, T.I. Oren, and B.P. Zeigler, Eds. North Holland, Amsterdam, pp. 165-179.

Overstreet, C.M., R.E. Nance, O. Balci, and L.F. Barger (1986) "Specification Languages: Understanding Their Role in Simulation Model Development," Technical Report SRC-87-001, Department of Computer Science, Virginia Tech, Blacksburg, Va., Dec.

Palme, J. (1976), "The Class Concept in the Simula Programming Language," FOA Report C 10052-M3(E5), National Defense Research Institute, Stockholm, Sweden, Aug.

Patrick, D.J. (1987), "The Applicability of Structured Modeling to Discrete Event Simulation Systems," Master's Thesis, Naval Postgraduate School, Monterey, Calif.,(Mar.).

Peterson, J.L. (1977), "Petri Nets," *Computing Surveys 9*, 3 (Sept.), 223-252.

Pidd, M. (1984), *Computer Simulation in Management Science*, John Wiley and Sons, New York.

Roberts, N., D.F. Andersen, R.M. Deal, M.S. Garet, and W.A. Shaffer (1983), *Introduction to Computer Simulation: A Systems Dynamics Modeling Approach*, Addison-Wesley, Reading, Mass.

Saydam, T. (1985), "Process-Oriented Simulation Languages," *Simuletter 16*, 2 (Apr.), 8-13.

Schriber, T.J. (1974), *Simulation Using GPSS*, John Wiley and Sons, New York.

Schruben, L. (1983), "Simulation Modeling with Event Graphs," *Communications of the ACM 26*, 11 (Nov.), 957-963.

Shannon, R.E. (1975), *Systems Simulation, The Art and Science*, Prentice-Hall, Englewood Cliffs, N.J.

Shub, C.M. (1978), "On the Relative Merits of Two Major Methodologies for Simulation Model Construction," In *Proceedings of the 1978 Winter Simulation Conference* (Miami Beach, Fla., Dec. 4-6). IEEE, Pisacataway, N.J., pp. 257-264.

Shub, C. M. (1980), "Discrete Event Simulation Languages," In *Proceedings of the 1980 Winter Simulation Conference 2* (Orlando, Fla., Dec. 3-5). IEEE, Piscataway, N.J., pp. 107-124.

Stevens, R.S. (1987), "A Tutorial on the Processing Graph Method," In *Proceedings of the 1987 Summer Computer Simulation Conference* (Montreal, Quebec, July 27-30). The Society for Computer Simulation, San Diego, Calif.

Teichroew, D. and J.F. Lubin (1966), "Computer Simulation- Discussion of the Technique and Comparison of Languages," *Communications of the ACM 9*, 10 (Oct.), 723-741.

Teichroew, D., F. Germano, and L. Silva (1983), "Applications of the Entity-Relationship Approach," In *Entity-Relationship Approach to Information Modeling and Analysis: Proceedings of the Second International Conference on Entity-Relationship Approach* (Washington, D.C., Oct. 12-14, 1981). North-Holland, Amsterdam, pp. 1-17.

Teichroew, D., P. Macasovic, E.A. Hershey III, and Y. Yamamoto (1980), "Application of the Entity-Relationship Approach to Information Processing Systems Modeling," In *Entity-Relationship Approach to Systems Analysis and Design: Proceedings of the International Conference on Entity-Retlationship Approach to Systems Analysis and Design* (Los Angeles, Calif., Dec. 10-12, 1979). North Holland, Amsterdam, pp. 15-38.

Tocher, K.D. (1963), *The Art of Simulation*, English Universities Press, London.

Tocher, K.D. (1965), "Review of Simulation Languages," *Operational Research Quarterly 16*, 2 (June), 189-217.

Tocher, K.D. (1966), "Some Techniques of Model Building," In *Proceedings of the IBM Scientific Computing Symposium on Simulation Models and Gaming* (Thomas J. Watson Research Center, N.Y., Dec. 7-9, 1964). IBM Data Processing Division, White Plains, N.Y., pp. 119-155.

Tocher, K.D. (1979), "Keynote Address," In *Proceedings of the 1979 Winter Simulation Conference* (San Diego, Calif., Dec. 3-5). IEEE, Piscataway, N.J., pp. 640-654.

Tocher, K.D. and D.G. Owen (1961), "The Automatic Programming of Simulations," In *Proceedings of the Second International Conference on Operational Research* (University of Aix-Marseille, Aix-en-Province, France, Sept. 5-9, 1960). John Wiley and Sons, New York, pp. 50-68.

Torn, A. A. (1981), "Simulation Graphs: A General Tool for Modeling Simulation Designs," *Simulation 37*, 6 (Dec.), 187-194.

Unger, Brian W. (1986), "Object Oriented Simulation - Ada, C++, Simula," In *Proceedings of the 1986 Winter Simulation Conference* (Washington, D.C., Dec. 8-10). IEEE, Piscataway, N.J., pp. 123-124.

Wasserman, K. (1984), "Understanding Hierarchically Structured Objects," Technical Report CUCS-124-85, Department of Computer Science, Columbia University, New York, N.Y., May.

Weitzman, E. (1986), "ECOS Tutorial (Draft)," Analytic Disciplines, Inc., Washington, D.C., May.

Zeigler, B.P. (1976), *Theory of Modelling and Simulation*, John Wiley and Sons, New York.

Zeigler, B.P. (1984a), "System-Theoretic Representation of Simulation Models," *IIE Transactions 16*, 1 (Mar.), 19-34.

Zeigler, B.P. (1984b), *Multifacetted Modelling and Discrete Event Simulation*, Academic Press, New York.

Zeigler, B.P. (1987), "Hierarchical, Modular Discrete-Event Modelling in an Object-Oriented Environment," *Simulation 49*, 5 (Nov.), 219-229.

VITA

Name: Emory Joseph Derrick

Address: 2719 Newton Court
Blacksburg, Va. 24060

Phone: 703-953-2000

Birthdate: 17 January 1952

Marital Status: Married, three children

Education:

M. S.- (candidate)
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061
1988

B.S. - Electrical Engineering
United States Naval Academy
Annapolis, Maryland
1974

From 1974 to 1980, Mr. Derrick served as an officer in the Submarine Service of the United States Navy. During the period 1981 to 1985, Mr. Derrick was employed by the U.S. Navy as a civilian General Engineer with the Naval Sea Systems Command. He is currently a Commander (Select) in the United States Naval Reserve. Mr. Derrick has been employed as a teaching and research assistant by the Computer Science Department and the Systems Research Center of Virginia Tech since 1985. Mr. Derrick is a student member of the Association for Computing Machinery and The Society for Computer Simulation.

Emory Joseph Derrick