

BREADTH-FIRST ALGORITHM FOR
QUALITATIVE DISCRETE
EVENT SIMULATION

By

NITIN SATYANARAYAN AGRAWAL

Bachelor of Engineering

University of Mumbai

Mumbai, India

July, 1998

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2003

BREADTH-FIRST ALGORITHM FOR
QUALITATIVE DISCRETE
EVENT SIMULATION

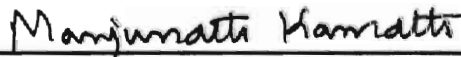
Thesis Approved:



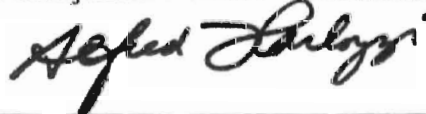
Dr. Ricki G. Ingalls - Chair



Dr. David B. Pratt – Committee Member



Dr. Manjunath Kamath – Committee Member



Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my advisor, Dr. Ricki G. Ingalls for his intelligent supervision, guidance, inspiration and support. I am grateful to him for all the extra time and effort he invested so that I could complete my thesis on time. My sincere thanks extend to my other committee members Dr. David B. Pratt and Dr. Manjunath Kamath for their suggestions, assistance and support.

I would like to thank Dr. William Kolarik and School of Industrial Engineering and Management for providing with this research opportunity. I would also like to give special thanks and love to my parents for their support and encouragement.

Finally, I would once again thank School of Industrial Engineering and Management for supporting during these years of study.

TABLE OF CONTENTS

Chapter	Page
1. Introduction.....	1
2. Literature Review.....	5
2.1. Classification of Simulation Models.....	5
2.1.1. Continuous-Time Simulation.....	6
2.1.2. Discrete-Event Simulation.....	7
2.2. Qualitative Simulation (QS).....	8
2.3. Qualitative Discrete-Event Simulation (QDES).....	13
2.3.1. Qualitative Discrete-Event Simulation Framework.....	16
2.3.2. Qualitative Discrete-Event Simulation Parameter Definitions.....	19
3. Research Methodology.....	23
3.1. Objective of Thesis.....	23
3.2. Scope and Limitations.....	24
3.3. Hypothesis.....	24
3.4. Thesis Phases.....	25
4. Depth-First Algorithm Review.....	27
4.1. Depth-First Methodology.....	27
4.2. Output of Depth-First.....	30
4.3. Depth-First Methodology Explained With Example.....	30
5. Breadth-First Algorithm Design And Implementation.....	36
5.1. Breadth-First Approach.....	36
5.1.1. Implementation Approach.....	37
5.1.2. Designing Steps for Breadth-First Algorithm Approach.....	38
5.2. Validating With An Existing Example.....	41
5.3. Breadth-First Algorithm Implementation.....	41
5.4. Explanation Of The Breadth-First Algorithm With An Example.....	52
5.5. Validating The Output.....	58

5.6. Run-Time Comparison.....	58
5.7. Additional Advantages Of The Breadth-First Algorithm	61
6. Summary and Future Research	63
6.1. Research Summary	63
6.2. Future Research	64
References.....	66
Appendix A. Interval Math.....	68
Appendix B. PERT Network Example	69

LIST OF FIGURES

Figure	Page
Figure 2.1: Event Graph With a Scheduling Edge and an Execution Condition (Ingalls, 1999).....	16
Figure 2.2: Event Graph for Single Machine Example (Ingalls, 1999).....	17
Figure 4.1: Execution Tree.....	31
Figure 5.1: Procedures for Execution of QSGM Model.....	42
Figure 5.2: Flow chart for Breadth-First Algorithm.....	47-49
Figure 5.3: Run Time Function for Breadth-First Algorithm.....	60
Figure B.1. PERT Network Event Graph.....	69

LIST OF TABLES

Table	Page
Table 5.1: Simulation clock time and state variables at some point in simulation.....	56
Table 5.2: Partial output for single machine example using breadth-first algorithm	57
Table 5.3: Run Time (Seconds) for Single Machine Example ($E = 8$)	60
Table B.1: The Depth-First Algorithm Partial Output for PERT Network Example	70
Table B.2: The Breadth-First Algorithm Partial Output for PERT Network Example	71

Chapter 1

Introduction

Simulation is the modeling of processes and operations of real-world systems over time. Simulation generates artificial data to predict the system's behavior without actually working with the real-world system. The system can be studied and analyzed using the data generated by simulation. "Simulation is the promotion of idea that process whose complete models are unknown can still be used as basis for computation," (Hocaoglu, 2003).

Simulation models are a representation of the actual system. These models require information about the parameters or variables of the system that are used to model the actual system. Traditional discrete-event simulation often uses probability distributions to describe these parameters. The probability distributions are based on certain assumptions by the modeler. Exact information about model parameters, such as the type of statistical distribution, is often not available. Although it is standard practice to make assumptions for these inputs in traditional discrete-event simulation, qualitative discrete-event simulation has created the constructs to define model parameters qualitatively. For example, if a customer arrives at a teller between 10:00 AM and 10:15 AM, then traditional discrete-event simulation cannot be used for modeling without making certain assumptions about the arrival time distribution for the customer. Traditional discrete-event simulation might assume that the value of time when the customer arrives at the teller is a random output from a uniform distribution with parameters of 10:00 AM and

10:15 AM. It is clear that the uniform distribution for arrival time is an assumption. The output obtained may not represent the true behavior of the system if the uniform distribution assumption does not hold. This might lead to faulty analysis of the system. Qualitative Discrete-Event Simulation (QDES) can be used to create models with fewer modeling assumptions.

Simulation models are largely classified into two types depending on how time is incremented. The two types are discrete-event simulation (DES) and continuous simulation (CS). Discrete-event simulation is defined as one in which the state variables change when an event occurs. In continuous simulation, the state variables change at defined time intervals. Continuous simulation models are described by using a set of differential or difference equations.

Qualitative Continuous Simulation (QCS) was first developed by Kuipers (1986). Kuipers described QCS models based on qualitative differential equations (QDE). According to Kuipers (1986), "Qualitative simulation systems produce the set of possible behaviors by generating and filtering the set of possible transitions from one qualitative state description to its successors. QCS is based on qualitative differential equation model in which variables are continuously differentiable functions. The range of each variable is defined qualitatively and it is a finite set of values on the real number line." QCS models start with initial values for the parameters. Successive states are derived continuously until the simulation terminates or all the possible behavior of the models are generated.

Qualitative Simulation Graph Models (QSGM) were developed by Ingalls (1999). QSGM is an alternative approach to the problem of Qualitative Discrete-Event Simulation (QDES). By combining discrete-event simulation with qualitative simulation,

QDES is able to model discrete event systems where exact information is not available or cannot be adequately quantified.

A QDES model uses imprecise specification of parameters such as the event occurrence time and the state variables. The event occurrence time is represented using an interval on the real-number line. When time is defined in real-valued intervals, it is normal for the order of events that are scheduled to be executed to be uncertain. For example, if the event occurrence time for one event is $[3,5]$ and the event occurrence time for a second event is $[4,6]$, the order of execution for these two events is uncertain. When this uncertainty exists, then the QDES creates a “branch” or a “thread” for each possibility. In the first thread, it is assumed that the event whose event occurrence time is $[3,5]$ is executed first. In the second thread, it is assumed that the event whose event occurrence time is $[4,6]$ is executed first. An individual thread is terminated when a specified condition is met or when no additional events are scheduled to occur. The simulation stops when all threads terminate.

The thread generation process generates a tree-like structure whose nodes are represented by events. An algorithm that uses depth-first traversal to generate all possible threads of the model has been developed by Ingalls (1999). The objective of this thesis is to develop a breadth-first traversal of the threads so that all active threads can be evaluated simultaneously.

In depth-first algorithms, the root node is determined first, then the child node of the root is determined, then the grandchild is determined, and so on. The generation of children continues on a single thread until the thread reaches the stopping condition. During the process, sibling nodes are placed in a stack to be executed later. In breadth-

first traversal, all of the sibling nodes are determined and placed in a queue. Then each sibling is taken out of the queue and executed. When a sibling node is executed, its children are placed in the queue. Roughly speaking, the execution goes from one level of the tree to another. This thesis proposes a breadth-first algorithm whose execution queue is managed in such a way that the simulation clock time is nearly equal for all of the nodes in the queue.

This thesis is organized into 6 chapters. Chapter 2 gives an overview of literature in the field of qualitative simulation. Chapter 2 also introduces to the concept of qualitative discrete-event simulation that was developed by Ingalls (1999). Chapter 2 concludes by defining the objective, purpose and scope of the thesis.

Chapter 3 describes the hypothesis and different phases of the thesis. Chapter 4 discusses the depth-first methodology developed by Ingalls (1999) and the breadth-first methodology that is proposed. Chapter 5 demonstrates the breadth-first algorithm implementation and provides an in depth explanation of the newly developed algorithm with the validation of the output. Chapter 6 summarizes the thesis and provides an insight into future research that could be done in the field of QSGM.

Chapter 2

Literature Review

Most real-world systems that change with time are so complex that they cannot be modeled mathematically. However, most of these systems can be modeled using simulation. According to Banks (1998), simulation is used to describe and analyze the behavior of a system. Simulations models help analyze the design of real-world operations and processes without building actual systems. This allows an analyst to answer what-if questions about the real system. Simulation models also help determine constraints and problems that could be faced by real-world systems before the actual system is in place, thereby saving a considerable amount of time and money. Efforts can be directed to solve the problems and to overcome the constraint during the system design phase. Simulation studies or models can be built for both existing and non-existing systems. Simulation models are widely used in manufacturing systems, queuing system, scheduling, material handling systems, capital investments decision making, cash flow analysis and supply chain modeling. Numerous applications of simulation in different fields make it a powerful modeling tool.

2.1. Classification of Simulation Models

Cellier (1991) have defined three types of mathematical models, which are:

- Continuous-time models
- Discrete-time models
- Discrete-event models

In continuous-time models, the state variables change their values continuously with time. Continuous-time models are represented using a set of differential equations for the variables that are differentiable with time. Conceptually, time is an analog variable and the simulation clock is advanced in sufficiently small steps in such a way that continuous time is approximated.

Discrete-time models are represented through a set of difference equations. In discrete-time simulation, the time is divided into discrete time steps and simulation clock is advanced by a fixed clock increment that is sufficiently large to make it noteworthy.

Discrete-event models change the state variables values only when something significant has occurred. As in continuous-time models, time is a continuous variable. What differentiates discrete-event models from continuous time models is the assumption that nothing significant occurs between two events.

Similarly, simulation can also be classified in two broad categories based on above distinction of mathematical models, which are:

1. Continuous-time simulation.
2. Discrete-event simulation.

2.1.1. Continuous-Time Simulation

In continuous-time models, the state of the simulation model is defined by dependent variables that change their values continuously over time. In Banks (1998), the state variables of continuous-time simulation models are represented in one of the following three ways:

- Functional form, in which the state variable is represented as a function of time and other system variable. For example, $x = f(y, t)$.

- Difference equations in which the state variable is represented as a difference in values from one time unit, t to next time unit, $t+1$. For example, $x_{t+1} = a x_t + b y_t$.
- Differential equations. For example, $dx/dt = f(y, t)$.

The state variables in a continuous-time simulation model are dependent on the time. In continuous-time models, time is typically considered as an independent variable, which is represented as t in above examples. Neelamkavil (1987) states that the simulation of a continuous system generates one or more numerical solutions which satisfy the differential equations defining the model for given initial condition using standard numerical method. These solutions satisfy the differential equations that define the model.. The initial values of the state variables are initialized at the starting point in time. These values are used as inputs to the differential equations which determine a new set of values when the simulation progresses to next point in time, that satisfy the set of equations, using numerical analysis procedure. Banks (1998) attributes the complexity in continuous time simulation models to following reasons:

- Randomness involved in the variables used to define the equations.
- Changes occurring in the equations used to define the models due to the continuous change of the equations coefficients.

2.1.2. Discrete-Event Simulation

In discrete-event simulation, variables change their values only when an event occurs. Discrete-event simulation models are both stochastic and dynamic in nature. Discrete-event simulation captures dynamic system behavior by evaluating how the entities and the activities in the simulation interact with each other. For example, in a single-server

system where a server serves a customer, the entity is the customer and one of the activities is the customer being served by the server. Events occur at the beginning and completion of each activity. In our example, an event occurs when the customer starts the service that is performed by the server. The next event occurs when the customer leaves the system when the service is complete. The state of the customer remains unchanged between start and end of the customer service by the server. The simulation clock advances at each event. When service begins, the simulation time is set to the time when the service is scheduled to begin. When the service ends, the simulation time is advanced to the time when the service is scheduled to end.

In cases where good quantitative information exists, quantitative analysis methods are most appropriate and efficient to study and analyze the models. However, if good quantitative information is not available or information is incomplete, then qualitative simulation may be a better methodology for modeling and analyzing the systems under study.

2.2. Qualitative Simulation (QS)

“Simulation solves problems executing their model on computers using numeric information, but QS uses simulation’s model execution approach for reasoning task”, (Hocaoglu, 2003). QS is a reasoning technique which solves problems by deriving useful inferences from models having considerably less information than is usually required to analytically solve the problem.

Hamscher et al. (1995) uses “boiling of water on stove” as an example to explain the QS technique. To write a program that could predict the behavior of the “boiling water” system, one would write a computer program to solve a set of differential

equations that would explain the relationship between the temperature of water, volume of water, specific heat of water, burner temperature, heat transfer coefficient, temperature of the air, height of pot above sea level, and other parameters. Traditional continuous simulation could have been used to analyze this model if the modeler could specify the exact form of functions explaining the relationship between the model parameters, the precise value of the parameters in those functions and the initial values of the variables. Traditional simulation would result in a solution that would explain the behavior of the system. However, there are times when the modeler does not know about the precise nature of the equations. Also, there is a question of which parameters need to be included and which parameters can be excluded from the model. For example, the modeler may not want to include the altitude of the pot. Also, the exact values of the initial conditions such as the temperature of the air and the temperature of water may not be known. However, qualitative information about the parameters may be available. An example of qualitative information for the boiling water example is given below:

1. The burner temperature is greater than boiling point of water.
2. The initial temperature of water is between 0°C and 100°C and the temperature of water is increasing.

QS can be used to predict the behavior of such systems. The three different behaviors of this system would be water is heated to boiling point from time 0 to some time t_1 , water is boiling from time t_1 to time t_2 and finally there is no water from time t_2 to infinity. The example does suggest some important properties of QS. First, it can work with less precise information. Second, it does not assume precise values of the variables to solve the model as compared to traditional simulation models.

One of the early pioneers of QS is Kuipers (1986). Most of his work is based on the qualitative differential equation model (QDE). “QDE is an abstraction of an ordinary differential equation, consisting of a set of real-valued variables and functional, algebraic and differential constraints among them. QDE model is qualitative because the values of variables are described in terms of their ordinal relations with a finite set of symbolic landmark values, rather than in terms real numbers and functional relations are described as increasing or decreasing over particular ranges, rather than specifying it in functional form.” (Kuipers, 2001). The QS technique described by Kuipers (1986) is used to solve continuous time models and it is referred to as Qualitative Continuous Simulation (QCS) in this thesis.

QCS generates all the possible behaviors of the system. This gives the decision maker the ability to choose from multiple options available to him for decision making. QCS described by Kuipers (2001) starts with a qualitative model and a qualitative description of the initial state. QCS uses an interval in the set of real numbers to define qualitative state variables. Kuipers (1986) describes the following inputs to the QCS algorithm using the “boiling water example”:

- A set of functions in the system. For example, a function describing relationship between temperature of the water and the burner.
- A set of constraints applied to the function variables. For example, “change in water temperature” is a derivative of “change in burner temperature” over time.

- An ordered set of symbols representing landmark values associated with each function. For example, temperature of water varies in the range of 0°C to infinite.
- The initial conditions at time t_0 for all functions and variables. For example, initial temperature of water is initialized between 0°C and boiling point of water. The temperature of the water is also increasing.

With these input values, the possible direct successors of the current state descriptions can be predicted using the qualitative description of the current state. The process is repeated to produce a graph that describes all of the qualitative states of the system. The result of the QCS is one or more qualitative behaviors for the functions and symbols. The qualitative behavior of the boiling water model consists of the following:

- Sequences of distinguished time points of the systems behavior. For example, the temperature of water is increasing at $t_0 = 0$, water starts boiling at time t_1 , and water evaporates completely at some time t_2 .
- Qualitative state description of the system between adjacent time points for each function and variables. For example, between time t_1 and t_2 , the temperature of water is increasing and is between boiling point of water and infinity.

The QSIM software is developed by Kuipers (1986) for executing QCS models represented using differential equations. Farquhar et al. (1994) has prepared the manual for the QSIM tool. The QSIM software for solving qualitative continuous time models uses all of the traits of the QCS algorithm developed by Kuipers (1986). The software also compares alternative approaches that are produced. The algorithm starts with a set of

constraints abstracted from a set of differential equations. Kuipers (1986) proved that the QSIM algorithm produces a qualitative behavior corresponding to any solution that would have been produced by the original set of differential equations. He also demonstrated that the qualitative simulation algorithm might produce spurious qualitative behaviors which do not correspond to any feasible output of the original set of differential equations.

QSIM executes the QCS model by deriving descendants of each qualitative state. The process of deriving descendants is repeated until all of the possible qualitative states are predicted. Kuipers (2001) suggested that the algorithm must ensure that all possible qualitative value transitions and their combinations are predicted. Also, combinations of qualitative values are deleted when they are inconsistent with the feasible output of the original set of differential equations.

Qualitative continuous simulation using qualitative differential equations for modeling physical systems has been developed and is practically applied to modeling physical systems. Wyatt et al. (1995) compared qualitative and quantitative simulation using a case study model of the interactive markets for housing and mortgages. They showed that the data or information required for qualitative simulation is considerably less in comparison to the data required for the quantitative simulation. Also, they claimed that quantitative simulation tends to hide some of the true behavior of the system by making invalid and impractical assumptions. One such assumption is about the interest rate. The interest rate for the mortgages is kept constant in the quantitative simulation model to keep model simple. This assumption in model will certainly affect the realism as the interest rate keeps on changing thereby hiding the true behavior of the system.

A combination of qualitative and quantitative simulation using numeric intervals to represent incomplete quantitative information is suggested by Berelant et al. (1997). They demonstrated that the combination would overcome the shortcomings of qualitative simulation by using the strengths of both techniques. All these studies focused on continuous models and did not consider discrete-event models.

2.3. Qualitative Discrete-Event Simulation (QDES)

To solve and analyze discrete-event simulation models with qualitative parameters, another approach to QS was developed by Ingalls (1999) that combines QS with discrete-event simulation. Ingalls defines the Qualitative Simulation Graph Model (QSGM), which is implemented with Qualitative Event Graphs (QEGs). QSGM is an alternative approach to the problem of Qualitative Discrete-Event Simulation (QDES). Bulitko et al. (2003) presented an alternative methodology to support a qualitative simulation of temporal concurrent processes using Time Interval Petri Nets. The methodology is similar to one defined by Ingalls (1999). It uses time intervals to represent uncertainty in inputs and outputs, similar to temporal intervals defined by Ingalls (1999). The next section briefly describes QDES methodology.

Ingalls (1999) extends the application of DES to systems for which accurate quantitative information is missing by introducing the qualitative description of time, delays, and state variables. Event execution time is defined using intervals in set of real numbers called temporal intervals. Two types of temporal interval defined by Ingalls (1999) are:

- Constant Intervals: “A constant interval is an interval whose value must be the same throughout the entire thread of the simulation, i.e. it is assumed that the actual values of the variable is a constant that lies somewhere in an interval.”
- Uncertain Intervals: “An uncertain interval is an interval whose value could be different every time that the interval is evaluated.”

For example, in a “single server system,” the arrival time and service time are described as any value in the range of some time t_1 and t_2 . These temporal intervals are modeled as *uncertain intervals* and their representation is $[t_1, t_2]$. During the execution of a traditional DES, a random sample would be taken that would be in this interval. The type of distribution defined for representing the arrival time is based on the assumptions made by the modeler, which may include fitting a distribution to past data. The assumption of the statistical distribution on the interval is not necessary with QDES, which allows the modeler to model and analyze systems with fewer modeling assumptions.

QDES approach differs from the QS models defined by Kuipers (1986) in following ways:

- Definition of the model – QDES is implemented using the qualitative simulation graph methodology (QSGM) while QCS uses qualitative differential equations.
- QDES is targeted to solve models that come under the umbrella of discrete-event models while QCS are used to solve physical models based on continuous-time models.

The execution approach followed by both QDES and QCS models are similar as both methodologies proceed by predicting possible direct successors of the current state. The process is repeated to produce a graph that describes all qualitative states of the system. Each path starting from the root gives all possible qualitative behaviors of the system.

QSGM generates threads, which are also called envisionments that characterize all possible behaviors. “Coverage is an important advantage of the QDES approach because it does not miss outcomes that a sampling based approach like traditional DES might with a finite sample size.” (Ingalls, 1999) The generation of threads increases exponentially with the complexity of the model being executed. This exponential explosion of threads creates a run-time issue with the algorithm on large models. This issue is considered a key research topic by Ingalls.

Ingalls (1999) implements thread generation using a depth-first algorithm that completely finishes one thread while putting additional threads on a stack to be executed at a later time. This approach is very efficient in the case where all threads need to be executed. However, Ingalls envisions situations where criteria could be included in a model that would differentiate threads by some objective. The depth-first generation of threads is not efficient in the case of an objective that requires a comparison of all active threads. In order to accomplish this goal, a breadth-first algorithm for thread generation and simulation execution would need to be developed.

The breadth-first algorithm described in this thesis provides an opportunity to evaluate the threads simultaneously and eliminate “unimportant” threads. Eliminating some threads may reduce the run time and thereby allow modelers to solve more complex

models. The breadth-first algorithm will enable researchers to solve more realistic and complex models and help to further develop QDES methodology.

2.3.1. Qualitative Discrete-Event Simulation Framework

Modeling methodology based on combination of Event Graphs and qualitative Simulation, called the Qualitative Simulation Graph Methodology (QSGM), is used to implement qualitative discrete-event simulation. QSGM uses the event graph construct to define QDES models. Figure 2.1 shows event graphs construct with a scheduling edge and an edge execution condition.

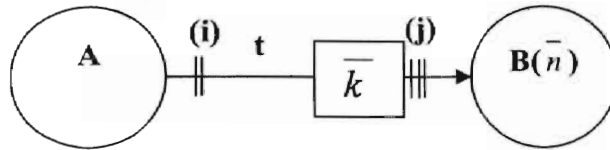


Figure 2.1: Event Graph with a scheduling edge and an execution edge condition, (Ingalls, 1999).

Events A and B are represented as nodes and edge connecting nodes indicates the relationship between the two events. The event graph framework shown in Figure 2.1 illustrates that if event A occurs and scheduling condition (i) is true at that instant, then event B will be scheduled to occur t time units later. If edge execution condition (j) is true t time units later then event B will be executed with the state variable array \bar{n} set equal to values in array \bar{k} (Ingalls, 1999).

As an example of a QSGM, consider the example of a single machine queuing system. In this example, when the job arrives at the machine, if the machine is idle then the machine starts processing the job immediately. Otherwise, the job joins the queue and waits for the machine to become available. When the machine becomes available, the job is delayed for the machining time. Upon completion of the job, the machine is made

available for another job. In this example, the buffer capacity is assumed to be infinite. The event graph for single machine example is shown in Figure 2.2.

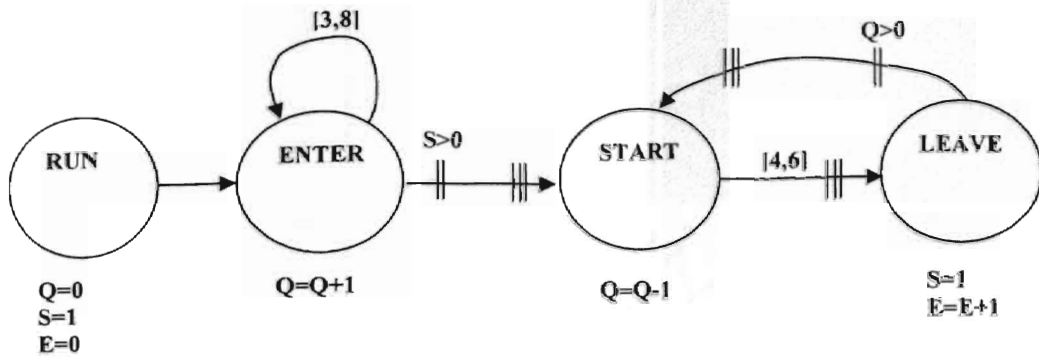


Figure 2.2: Event Graph for single machine example, Ingalls (1999).

The nodes RUN, ENTER, START and LEAVE are events which represent the following:

RUN – the starting event that starts the simulation.

ENTER – the arrival of the job in single machine queue system.

START – the job starts its processing at the machine.

LEAVE – the job completes its processing and exits.

The state variables are defined as:

Q – the number of jobs in the queue waiting for processing at the machine,

S – the number of machines available for serving customers. For a single machine system S can be equal to either 0, when the machine is busy, or 1, when machine is idle,

E – the number of jobs that have been processed and left the system.

For this example, all of the edge execution conditions are *TRUE*. The scheduling conditions are represented on the edge connecting two nodes. In Figure 2.2, the scheduling conditions are $S > 0$ and $Q > 0$. When the ENTER event occurs, then it will

schedule the START event without any delay if the scheduling condition $S > 0$ is true. The condition $S > 0$ is true if at least one server is available. Similarly, the scheduling condition $Q > 0$ represents that the START event will be scheduled without any delay only if $Q > 0$ when LEAVE event occurs. The condition $Q > 0$ is true if there are jobs in queue waiting for machine to become available.

Temporal intervals are used for the edge delay times. The interval $[3,8]$ on the ENTER-ENTER edge represents the inter-arrival time between jobs. In this case, the inter-arrival time between two jobs can be anywhere between 3 and 8. Similarly, the interval $[4,6]$ on the START-LEAVE edge indicates that the job completes its processing on the machine after a delay of at least 4 and no more than 6.

Below each node is a set of equations that are used to evaluate the state change variables. When the RUN event occurs, then the state variables are initialized to $Q=0$, $S=1$, and $E=0$. When an ENTER event occurs, then the number of jobs in the queue increases by 1 ($Q=Q+1$). Similarly, when a new job starts its processing at the machine, then the Q value is reduced by 1 ($Q=Q-1$) and the state of the machine is changed from idle ($S=1$) to busy ($S=0$). When the job leaves the system after being processed then the machine state is changed from busy to idle and the number of units that have exited the system is increased by 1 ($E=E+1$).

The QSGM framework helps define the real-world system using the above set of notations and modeling approach. The next section will discuss the execution of the QSGM model using the framework defined above.

2.3.2. Qualitative Discrete-Event Simulation Parameter Definitions

Banks (1998) explains that a discrete system model consists of some or all of the following:

1. Model – Representing a real-world system.
2. Event – Occurrence of an event that changes the state of the system.
3. System State Variables – A collection of variables that represent or are used to define what is happening in the system.
4. Entities – Entities represent objects that move through the system.
5. Attributes – Entities may have some values associated with them called attributes.
6. Resources – Entities are served by the resources.
7. Current Event Calendar – A list that represents events that are scheduled to occur at the current time of the simulation.
8. Future Event Calendar – A list that represents the events that are scheduled to occur at some time in future.
9. Simulation Clock – Represents the current time of the simulation. The clock time is advanced to the future time when an event is scheduled and the state change of the system occurs.

Since QSGM is a derivation of the Simulation Graph Methodology (SGM) introduced by Yücesen and Schruben (1992), then QSGM does not have all of the constructs defined by Banks. In particular, QSGM does not have entities and resources. Also, attributes are defined as a list of values passed from one event to another.

Similar to traditional DES, the simulation clock time represents the current clock time of the simulation and is a real-valued variable. The future events calendar forms a

sorted list of time-delayed events. The time-delayed events will not become active until some future simulated time is reached. The first event on the future events list is event that will occur next.

In QSGM, the simulation clock time is not represented as a real-valued variable, but as a temporal interval. The future events calendar used in QSGM is a single list that stores *event notices*. An event notice stores relevant information about the event that is to be executed. The event notices are sorted by a mechanism that assures that the first event notice on the calendar can possibly be the first event to be executed on the current thread. (Ingalls, 1999) Each event notice has following set of values:

1. The time when event is to be executed.
2. The execution priority for event. (The priority of an event notice helps in breaking a tie when two event notices have equal execution time.)
3. The node (event) to be executed.
4. The edge that scheduled the event notice.
5. The values of the edge attributes.

An example of an event notice is $([0,0], 1, START, ENTER-START, TRUE)$. It states that START event is scheduled to occur at time $[0,0]$ and was scheduled by the ENTER-START edge. The edge execution condition for the event notice is *TRUE*. The START event is executed when this event notice is removed from the future events calendar.

As in DES, the QSGM future event calendar or list is also sorted in order of event occurrence time, but the rules used to sort the list are based on interval math (Appendix A). In QSGM, if the time for event notices 1 and 2 are defined as $[t_1^-, t_1^+]$ and $[t_2^-, t_2^+]$,

then the following conditions are evaluated in order to determine if t_1 precedes t_2 in the future events calendar:

1. $t_1^+ < t_2^-$
2. $t_1^- < t_2^-$
3. $t_1^- = t_2^-$ and $t_1^+ < t_2^+$

In traditional discrete-event simulation, if there is more than one event on the current events calendar, then the order of execution is determined arbitrarily based on the type of algorithm used. In the QSGM, if two or more event notices could possibly be executed next, then QSGM generates a set of threads for all possible combinations of event sequences. For example, if the two events A and B , with equal priority, are to occur at time $[t_a^-, t_a^+]$ and $[t_b^-, t_b^+]$, and it is not possible to determine if event A occurs first or event B occurs first, then QDES will generate two threads. One will execute event A first and the other will execute event B first. Ingalls (1999) defined the situation where the execution order of multiple events is uncertain as a non-deterministically ordered set (NOS). The members of the NOS are called non-deterministically ordered events (NOEs). Each NOE becomes the next event to be executed a new thread. These threads, which comprise all possible event orderings, allow QSGM to characterize all possible behaviors of the system, which is the essence of QS.

QSGM execution is similar to traditional DES. A QSGM model is executed over time by the mechanism that moves the simulated time forward. Since a temporal interval is used to define time in a QDES model, interval math (Appendix A) is used to calculate event times for the event notices in the future events calendar.

Ingalls (1999) has also included the qualitative specification of state variables such as the number of servers available. For example, if number of servers available in the system at any given time is between 2 and 5 then it is stated as $[2, 5]$. The qualitative specification of the state variables will not be considered in this thesis for simplicity.

Chapter 3

Research Methodology

3.1. Objective of Thesis

The objective of this thesis is to make a contribution to the field of qualitative discrete-event simulation by developing a breadth-first algorithm for the Qualitative Simulation Graph Methodology. An algorithm for solving qualitative discrete-event models using breadth-first traversal methodology is implemented using the object-oriented programming language, C++. Different models are executed using the algorithm to check its validity and the output is verified with output generated by depth-first algorithm presented by Ingalls (1999) in his dissertation.

With QSGM, Ingalls (1999) developed a simulation tool for solving discrete-event models for which precise information is not available. However, the algorithm developed by Ingalls (1999) uses depth-first traversal for solving QSGM models. This thesis assumes that the model could also be solved using breadth-first traversal and utilizes the advantages of breadth-first traversal algorithms. The depth-first algorithm does not allow the modeler to analyze all the threads simultaneously as each thread is executed until it reaches the stopping condition criteria. At any time during simulation run, the depth-first algorithm can only depict the partial behavior of the system. This is so because only certain threads are completely executed while other threads are waiting on the stack or may not even have been determined. The breadth-first algorithm provides a solution to the above problem.

3.2. Scope and Limitations

The breadth-first algorithm developed will only considers uncertain intervals for describing time delays in the system. Even though state variables can be defined qualitatively, this thesis does not consider the qualitative definition of state variables. The conceptual use of the breadth-first algorithm is discussed using hypothetical examples only.

3.3. Hypothesis

As mentioned earlier, Ingalls (1999) developed the Qualitative Simulation Graph Methodology (QSGM), as modeling framework for QDES. Ingalls (1999) developed a depth-first algorithm for QDES execution that first executes the root node, then the children of the root node, followed by the grandchildren of the root node, and so on until the thread is complete. After the thread is executed, sibling nodes are executed until the thread that they created is complete. Since the depth-first traversal of the nodes is possible, the hypothesis is to show that breadth-first traversal of the nodes is also possible.

Since QSGM is still in its very early stages, a set of algorithms and analysis techniques that would make QSGM a useable tool for modelers is still being developed. A breadth-first algorithm enhances QSGM so that it is more practical for modelers, especially in the area of strategic decision-making.

The algorithm would allow modelers to inspect all threads that are active at a particular time in the simulation and depending on their states. It would be possible to terminate the simulation of threads that are less meaningful to the decision maker. By removing less meaningful threads, the run-time of the algorithm can be reduced.

Developing the breadth-first algorithm will require a data structure to store sibling nodes, which will be created dynamically. These dynamically created sibling nodes will represent all the possible nodes that can be executed. Each node corresponds to one thread that is active in the simulation. Each node will be stored in this data structure and be sorted with respect to the event execution time. The sorting of this data structure will ensure that all of the threads in the data structure have their execution time very close to each other.

The purpose of the newly developed breadth-first algorithm will be demonstrated using a hypothetical example. Arbitrary criteria will be used to eliminate threads using the breadth-first algorithm and the run time will be compared with the run time for depth-first algorithm.

3.4. Thesis Phases

The thesis is carried out in discrete phases with each phase making progress toward achieving the objective of the thesis.

Phase I: To Study Qualitative Discrete-Event Simulation

First phase of the thesis concentrates on understanding the concepts of QDES approach that was developed by Ingalls (1999). The study of QDES methodology focuses on understanding the Qualitative Simulation Graph Methodology modeling approach that is used to describe models for QDES. The depth-first algorithm is used for solving or executing the simulation.

Phase II: To Code the Depth-First Algorithm For QSGM

The depth-first algorithm developed by Ingalls (1999) is coded in Smalltalk. To make this thesis possible, the QSGM algorithm has to be re-coded in C++. This phase is

necessary as it will help to compare the output obtained from using the depth-first algorithm developed by Ingalls (1999) and proposed breadth-first algorithm.

Phase III: To Develop the Breadth-First Algorithm

Third phase focuses on developing the proposed breadth-first algorithm for solving or executing QDES models. The algorithm is coded in C++ using Visual Studio 6.0 environment. The developed code is tested with examples described by Ingalls (1999).

Phase IV: To Validate the Breadth-First Algorithm

The output of the proposed algorithm is validated by comparing the output from depth-first algorithm since the output from both the algorithms must yield the same results.

Phase V: To Show the Purpose Of The Breadth-First Algorithm Using An Example

A hypothetical example is used to show the purpose of developing the breadth-first algorithm. The example will limit the number of active threads in the breadth-first algorithm to a smaller number and reduce the run time for the algorithm.

Chapter 4

Depth-First Algorithm Review

The framework discussed in the Chapter 2 is based on the Qualitative Simulation Graph Methodology. With the framework and parameter definition, the depth-first approach for the QDES developed by Ingalls (1999) will be discussed using the “single machine system” example.

4.1. Depth-First Methodology

The execution of QSGM resembles traditional DES to some extent. Ingalls (1999) developed the depth-first algorithm to execute QSGM models. The following definitions of parameters are necessary to understand steps in the depth-first algorithm.

L – the future events calendar, which is an ordered set of event notices.

S – the set of state variables. In the single machine system example the state variables are S (the number of available servers), Q (the number of jobs waiting in the queue), and E (the number of jobs that have exited the system).

H – the set of saved states. A saved state consists of the global event calendar L and the state variable array S . This set is used to recurse through all possible states in the simulation. H is also known as the *stack*.

N_h – the non-deterministically ordered set (NOS). This set contains the event notices that can possibly be executed.

h – the number of saved sets in H .

n_h – the variable to iterate through the N_h set.

With the previously defined parameters, the steps in depth-first algorithm developed by Ingalls (1999) are as follows:

1. Initialize the saved state set and counter, $H = \emptyset$ and $h = 1$.
2. Initialize the global simulation clock to time $t = [0,0]$.
3. Insert one or more event notices into the event calendar, L , that could be executed at time $[0,0]$.
4. Determine the NOS N_h , the set of all event notices that could be executed next.
5. If the number of events that can be executed next equals to 1 ($|N_h| = 1$) then go to step 9 else go to step 6.
6. Initialize the variable to loop through N_h , $n_h = 1$.
7. Save the state of the simulation by saving the state variable information, S , and the future events calendar information, L , in the save-state stack, $H_h = \{S, L\}$ and increase the save-state counter, $h = h + 1$.
8. Set $I = (N_{h-1})_{n_{h-1}}$ and remove event notice I from the global event calendar L .
 $L = L \setminus \{I \mid I = (N_{h-1})_{n_{h-1}}\}$. Go to step 10.
9. Remove the first event notice I from the global event calendar L . Since the global events calendar is sorted according to the event execution time, the first event notice on the calendar is one that would be executed next.
10. If the execution edge condition evaluates to *FALSE* then go to step 16, else go to step 11.
11. Determine the possible new simulation clock time. The new simulation time is calculated as follows. Suppose current simulation time is represented by $[t_c]$,

t_c^+] and event time by $[t_e^-, t_e^+]$ than new simulation clock time t' would be $[\max(t_e^-, t_c^-), \min(t_e^+, \forall l \in L)]$.

12. Update the simulation clock time $t=t'$.
13. Assign attributes to appropriate state variables.
14. Evaluate the state change.
15. Schedule further events. Schedule all events that are connected to edges with the current event if the scheduling edge condition is *TRUE*. The scheduled events are the events that will occur after delay times that are represented by temporal intervals on top of respective edges. Assign all attribute values to the new event notice for the scheduled event with the event time calculated by adding delay time to the current simulation clock time using interval math (Appendix A).
16. Stop simulation of the thread if any of the following condition is true:
 - Simulation clock time exceeded simulation stop time defined by the modeler, or
 - The simulation stopping condition defined by the modeler is evaluated as *TRUE*, or
 - The global event calendar L is empty.

If the saved state stack is empty, which is shown by $h = 1$, then terminate the simulation.

17. Increment $n_{h-1} = n_{h-1} + 1$. If $n_{h-1} \leq |N_{h-1}|$ then go to step 8.
18. Restore the last saved system state values that have been stored in step 7. $h = h - 1$, $L = (L \mid L \text{ in set } H_h)$, $S = (S \mid S \text{ in set } H_h)$. Go to step 8.

The algorithm developed by Ingalls (1999) is discussed with the help of the single machine system example. The stopping condition defined for terminating the simulation is when number of jobs that have left the system equals 5. Simulation stop time is set to infinity, since we do not want to stop the simulation until the number of exits equals 5.

4.2. Output of Depth-First

The partial output for the single machine system is presented in a graphical tree form in Figure 4.1. Complete output is shown from Ingalls (1999).

The oval shapes in Figure 4.1 represent the nodes. First line of each node indicates the event executed and current simulation clock time. The lines following the first line inside the node show event notices in global events calendar. The event notices in Figure 4.1 represent the time when the event executes, its priority, the node that will be executed next, and value of state variables S , Q , and E . The edges that schedule the nodes are not shown in the output. The nodes are numbered arbitrarily and do not necessarily represent the sequence in which the nodes will be reached or executed.

4.3. Depth-First Methodology Explained With Example

The depth-first approach for the QSGM starts by initializing the model parameters or variables to initial values. Initially the saved state stack H is empty and the variable h

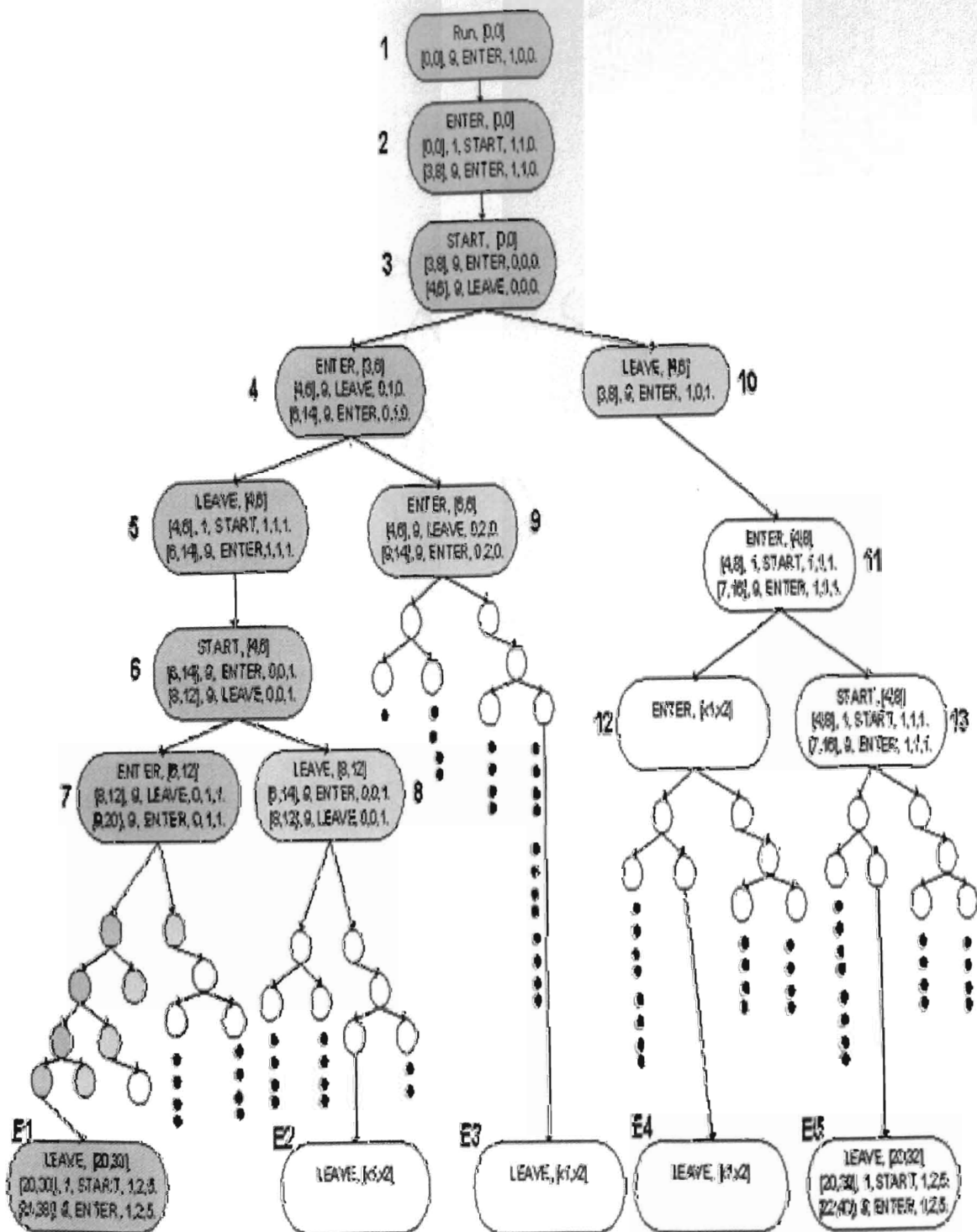


Figure 4.1: Execution Tree

is set equal to 1. In step 2, the simulation clock time is set to $[0,0]$. The first event scheduled to occur at time $[0,0]$ is a RUN event, which is inserted in the global events calendar L . In step 4, the NOS N_h will have only the RUN event notice as it is the only event that can be executed next. In step 5, since $|N_h| = 1$ the execution moves to step 9. In step 9, the RUN event notice is removed from L . RUN event does not have any edge execution condition and is always evaluated as *TRUE*. The simulation clock time is determined as $[0,0]$ in step 10 and updated. The RUN event updates the list of state variables to values, $S = 1$, $Q = 0$, and $E = 0$. The RUN event schedules an ENTER event because the edge scheduling condition is *TRUE* on the edge RUN-ENTER. Since the edge RUN-ENTER does not have any delay, the event execution time for ENTER is $[0,0]$ and the variables are given the values $S = 1$, $Q = 0$, and $E = 0$. The event notice inserted in the calendar is $([0,0], 9, ENTER, RUN-ENTER, TRUE)$. In step 16, simulation stopping condition evaluates to *FALSE*, since simulation clock time is less than stopping time, $E \neq 5$, and events calendar is not empty. Hence the stopping condition for this thread is false and simulation goes to step 4.

Since the ENTER event is the only event contained in the global event calendar, L , it forms the NOS and is the only event that can be executed next. The ENTER event notice is removed and the execution condition is evaluated as *TRUE*. The simulation clock time is updated to $[0,0]$. The state change is evaluated using the equation $Q = Q + 1$ and state variables are updated to $S = 1$, $Q = 1$, and $E = 0$. Two edges emanate from ENTER node and therefore two events are scheduled on the events calendar as follows:

- The START event, which is scheduled without any delay since the scheduling edge condition ($S > 0$) evaluates *TRUE* because $S = 1$. The event notice inserted in the calendar is $([0,0], 1, \text{START}, \text{ENTER-START}, \text{TRUE})$.
- The ENTER event (the edge ENTER-ENTER does not have any scheduling condition), is scheduled after a delay of $[3,8]$. The event time is calculated using interval math, which adds the delay time of $[3,8]$ to current simulation clock time of $[0,0]$ resulting in an event time of $[3,8]$. The event notice inserted in the event calendar is $([3,8], 9, \text{ENTER}, \text{ENTER-ENTER}, \text{TRUE})$.

In step 16, the stopping condition evaluates false and the execution moves back to step 4.

Next, the event START is executed, the simulation clock time t is updated to $[0,0]$ and the state change is evaluated using equation $Q = Q - 1$ and $S = 0$. This changes state variables to $S = 0$, $Q = 0$ and $E = 0$. The START event schedules a LEAVE event after a delay of $[4,6]$. The LEAVE event is scheduled to occur at time $[4,6]$ and now the events calendar has two notices on it:

- $([3,8], 9, \text{ENTER}, \text{ENTER-ENTER}, \text{TRUE})$
- $([4,6], 9, \text{LEAVE}, \text{START-LEAVE}, \text{TRUE})$

Step 16 evaluates stop condition to be false and a simulation execution goes to step 4.

At this point in step 4, it is impossible to determine the order of the events on the calendar because the execution times of the two event notices intersect with each other and the priorities of the two event notices are equal. The algorithm will create a thread for each event by executing each event first. Both events are in the set N_h and $|N_h| = 2$. Since $|N_h|$ is not equal to 1, the variable n_h is set to 1 and the execution goes to step 7.

Simulation state is saved in the saved-state stack, $H_l = \{S, L\}$ and the counter h is incremented to 2.

First, event notice ENTER is removed from the global events calendar L . The edge execution condition is *TRUE* and the new simulation time is calculated in step 11. Since the current simulation time is $[0,0]$ and the execution time of the events are $[3,8]$ and $[4,6]$, the simulation clock time set to $[\max(3,0), \min(8,6)] = [3,6]$. Since the ENTER event occurs before the LEAVE event in this particular thread, it is necessary that the ENTER event occurs in time $[3,6]$ since it cannot occur after the LEAVE event. State changes are evaluated and state variables are updated to $S = 0$, $Q = 1$, and $E = 0$. The ENTER event is scheduled to occur after a delay of $[3,8]$. The START event cannot be scheduled because the scheduling edge condition $S > 0$ is *FALSE*. The process continues until the stopping condition of $E = 5$ is *TRUE* for this thread. As the thread executes, the saved state stack is adding events that will be taken off of the stack and executed as new threads at a later time. Following the steps in the algorithm, the execution will reach the node labeled *E1* in Figure 4.1. The node labeled *E1* indicates end of thread 1. At the end of thread 1, the values of the state variables are $S = 1$, $Q = 2$ and $E = 5$.

As thread 1 progresses, there are several more events placed on the stack, H , and h is incremented accordingly. At some time in the future, these events will be taken off of the stack and h will be decremented until it is again set to 2. At that time, the LEAVE event that was placed on the stack is ready to be taken off.

At that time, Step 17 increments n_{h-1} by 1 and the value of $n_{h-1} = 2$ is equal to $|N_h|$. Since the saved state stack is not empty at this point, the simulation state is restored in step 18. Now, the current simulation time is be $[0,0]$, the state variables are $S = 0$, $Q = 0$

and $E = 0$. The global events calendar L will have the following event notices with the LEAVE event to be executed first (since ENTER event has already executed first in the previous thread):

- $([3,8], 9, ENTER, ENTER-ENTER, TRUE)$
- $([4,6], 9, LEAVE, START-LEAVE, TRUE)$

Simulation moves to step 8 where the LEAVE event is removed from the events calendar and execution continues until the terminating condition in step 16 is satisfied. At the end of the simulation all leaf nodes of the tree will have $E = 5$ and the saved state stack H will be empty.

This algorithm executes one thread until its stopping condition is reached and then the next thread is executed. The process goes on until all possible threads of the system are generated. Hence this algorithm is termed the “depth-first” traversal methodology.

Chapter 5

Breadth-First Algorithm Design And Implementation

5.1. Breadth-First Approach

The QSGM algorithm can also be implemented using a breadth-first traversal of the simulation nodes. A breadth-first traversal examines all firsts of the sibling trees before it examines any child tree (Weiss, 2000). Breadth-first traversal will cover all of the same threads that the depth-first traversal covered. The problem with the breadth-first traversal is that it cannot be defined recursively. The recursive nature of the depth-first algorithm gives it a speed advantage over breadth-first if all threads are generated. The breadth-first algorithm carries the overhead to continuously manage swapping in and out of nodes for all the active threads.

The depth-first algorithm is executed recursively until all of the threads are executed and the stopping condition is reached. In order to follow the breadth-first traversal, all the sibling nodes will be executed before traversing through the child nodes. This could help modelers to keep track of the each possible state and each possible option that is available. For example, in the single machine problem, the modeler may be interested in knowing the completion time of the first job under every possible thread or condition. In breadth-first algorithm, it will be possible to know the state of the simulation at each important landmark in all possible threads. This cannot be easily determined in the depth-first traversal. In the depth-first algorithm, the state of all of the

threads at certain point in simulation can only be determined only after the simulation has completed.

5.1.1. Implementation Approach

As with all recursive algorithms, the depth-first algorithm goes through two distinct phases. The first phase is building up the stack. A stack is built in the memory when the function calls itself. All variables are stored in a stack. The second phase is unwinding the stack. The stored functions on the stack are removed last-in first-out and the stack eventually becomes empty.

In the breadth-first algorithm, we propose to use a queue structure that will contain the nodes (representing the threads of the simulation) that are to be executed. The first node on the queue will be removed and executed. The execution of the node will schedule new nodes. The newly created nodes will be inserted in the queue and sorted according to the event execution time. The execution will be stopped when the queue is empty.

The breadth-first algorithm will only consider uncertain intervals and not constant intervals. "A constant interval is an interval whose value must be the same throughout the entire thread of the simulation, i.e. it is assumed that the actual value of the variable is a constant that lies somewhere in an interval," (Ingalls, 1999). Ingalls defined an uncertain interval as an interval whose value could be different every time that interval is evaluated. In order to keep the algorithm simple, constant intervals and qualitative definitions for state variables will not be considered.

5.1.2. Designing Steps for Breadth-First Algorithm Approach

The depth-first approach algorithm serves as a guideline for developing the logic for the breadth-first algorithm. Two more variables are defined for the breadth-first algorithm. These two variables are:

BreadthFirstNode – A set that consists of the event notice to be executed next, the state variable array, and the global events calendar to be used when the event notice is executed.

BreadthFirstNodeQueue – The queue used to store *BreadthFirstNodes*.

Also, two variables changed their definition for the breadth-first algorithm. The change is needed because the H set is no longer needed in the breadth-first algorithm. Those two variables are:

N – the non-deterministically ordered set (NOS). This set contains the event notices that can possibly be executed. Since H is no longer needed, there are not multiple instances of this set.

n – the variable to iterate through the N set.

The steps based of the breadth-first algorithm are as follows:

1. Initialize the simulation clock to time $t=[0,0]$. Initialize the state variables to their initial values in the state variable array S . Initialize the global events calendar L to be empty.
2. Create an empty queue to store the *BreadthFirstNodes*; call it the *BreadthFirstNodeQueue*.
3. Insert one or more event notices into the global events calendar, L , that could be executed at time $[0,0]$.

4. Create a *BreadthFirstNode* whose members are first event notice on the global events calendar L , the current simulation time t , the set of state variables S and the global events calendar L . Put it in the *BreadthFirstNodeQueue*.
5. Assign the current simulation clock time, t , to the simulation time stored in the first *BreadthFirstNode*, the state variables, S , to the values of the state variables stored in the first *BreadthFirstNode*, and the global events calendar, L , the global events calendar stored in the first *BreadthFirstNode*. Remove the first *BreadthFirstNode* from the *BreadthFirstNodeQueue*

Note: This step is equivalent to the step 18 of the depth-first algorithm where the state of the simulation is restored. In the breadth-first algorithm, the state of the simulation is restored every time a new node is removed from the *BreadthFirstNodeQueue*.

6. If the execution edge condition evaluates to *FALSE* then go to step 15, else go to step 7.
7. Determine the new simulation clock time t' . The new simulation time is calculated as follows: if current simulation time is represented by $[t_c^-, t_c^+]$ and event time by $[t_e^-, t_e^+]$ then the new simulation clock time $t' = [\max(t_e^-, t_c^-), \min(t_e^+ \forall l \in L)]$.
8. Update the simulation clock time with the time calculated in step 7. $t = t'$.
9. Assign attributes to the parameters of the vertex.
10. Evaluate the state change.
11. Schedule further events. Schedule all events that are connected with edges to the current event if the scheduling condition is *TRUE*. Assign any necessary

attribute values to the new event notice. Assign the event execution time by adding the delay time to the current simulation clock time, t , using interval math. (Appendix A)

12. Determine the NOS N , the set of all event notices that can be executed next from the global events calendar. Set $n = 1$.

13. If $n \leq |N|$ is *TRUE* then

for the event notice N_n , evaluate the simulation time t' when the event will scheduled to occur. Suppose the current simulation time is represented by $[t_c^-, t_c^+]$ and the event notice N_n event execution time by $[t_n^-, t_n^+]$, then the time which N_n is scheduled to occur is $t' = [\max(t_n^-, t_c^-), \min(t_n^+ \forall l \in L)]$.

else

go to step 15.

14. If any of the following conditions are true:

- The time which event N_n is schedule to occur, t' , has exceeded the simulation stopping time defined by modeler, or
- The simulation stopping condition defined by the modeler is evaluated as *TRUE*, or
- The global event calendar L is empty.

then go to step 15,

Else

Copy the global events calendar, L , to a temporary calendar, C . Remove event notice N_n from C . Create a *BreadthFirstNode* with reference to event notice N_n whose members are N_n , the time at which event N_n is

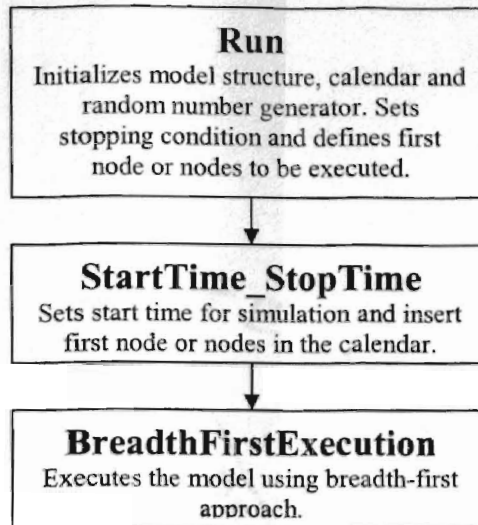


Figure 5.1: Procedures for Execution of QSGM Model

Before executing the algorithm, the user has to define the structure and the functions of the model. The structure consists of defining nodes and edges of the QSGM model and also the functions that are used when the events occur. The functions are responsible for updating state variables. The following variables have to be defined in order to describe the breadth-first algorithm.

globalCalendar – set of events that are scheduled to occur at some time in future.

The *globalCalendar* is set to be the calendar for the thread that is being executed.

event – the event notice being executed. Each event notice is a set whose elements are the time at which the event will occur, the priority of the event, the event that scheduled this event, the edge execution condition and the attributes that are passed to the event being executed. The variables used to define the set are:

event.time – temporal interval indicating time when the event is scheduled to be executed.

event.toMethod – event scheduled at *event.time*.

event.priority – priority of the event.

event.executionCondition – a conditional statement that evaluates either *TRUE* or *FALSE*. If it is *TRUE*, then event is executed.

event.attributes – attributes of the event.

globalCalendar.time – denotes current simulation time $[t^-, t^+]$ for the current thread.

stateVariableArray – array of all variables that represents state of the simulation.

In the single machine system example, the state variables are value of the server (*S*), the queue (*Q*) and the exits (*E*).

startTimeValue – time interval representing start time of the simulation.

stopTimeValue – time interval representing the stop time of the simulation. The simulation cannot go beyond *stopTimeValue*.

The time interval structure has variables *start* and *stop* representing the beginning and ending of the time interval respectively. For example, *startTimeValue.start* is the beginning of the *startTimeValue* interval.

The QSGM algorithm starts with the user initiating the execution of the model by calling the **Run** procedure. The **Run** procedure initializes model parameters and calls the **StartTime_StopTime** procedure that starts the execution of the simulation by inserting the first event or events into the calendar.

Procedure Run

Initialize the random number generator if required

Initialize the *globalCalendar*

Set the stopping conditions

Define the first node or nodes to be executed (can be overridden in model initialization)

Call *StartTime_StopTime()*

End Procedure Run

The **Run** procedure initializes the state variables of the model and defines the structure of the simulation model for execution. Also, the general structure that is required for any qualitative discrete-event model, like the random number generator (only if required) for the model and the global events calendar, is initialized. It also defines and stores the first event or events that are to be executed at the start of simulation. A call is made to procedure *StartTime_StopTime* that executes the QSGM algorithm.

Procedure *StartTime_StopTime*(*startTimeValue*, *stopTimeValue*)

Set *globalCalendar.time* = *startTimeValue*

Insert first node or nodes to be executed into the *globalCalendar*

Call ***BreadthFirstExecution*** (first event on the calendar to be executed, *startTimeValue*)

End Procedure *StartTime_StopTime*

The procedure *StartTime_StopTime* sets the *globalCalendar.time* to *startTimeValue*. The events that will be executed at the start of the simulation are inserted into the *globalCalendar*.

In the depth-first algorithm developed by Ingalls (1999), the saved state stack is used to store the simulation state as new threads are created. The state stack stores current values of state variables array and the global events calendar. In the breadth-first algorithm, threads are swapped in and out of execution so that the simulation clock time

of all of the threads is nearly equal. In order to manage the swapping of threads into and out of execution, a structure that holds certain information about each active thread is required. The structure is called the *BreadthFirstNode* structure and for each thread, it stores the global events calendar and the state variable array. *BreadthFirstNodes* are stored in the *BreadthFirstNodeQueue*. The elements of the *BreadthFirstNode* structure are:

The variables that are initialized to trace threads and events generated during the execution of models are as follows:

eventNumber – represents the number of events that have occurred.

spawningEvent – denotes the number of the event that spawned or created this thread.

modelThread – number of the model thread. Each thread has a unique number. The maximum value of *modelThread* represents the total number of threads in the model.

modelThreadEvents – denotes the number of events within a model thread.

The variables for storing the information required to execute the simulation of the thread are:

eventToBeExecuted – represents the event in the calendar that is scheduled to be executed when this thread is executed next. If the NOS that generated this *BreadthFirstNode* has more than one event, then there will be different *BreadthFirstNodes* that represent the alternative sequences of execution.

calendarAssociatedWithNode - represents the calendar that is associated with this thread.

stateVariableArray – stores the values of the state variables of the thread.

After initialization is complete, the procedure ***BreadthFirstExecution*** that executes the QSGM model using breadth-first traversal is called using two parameters, first event on the calendar and *startTimeValue*. The following variables are used in the breadth-first algorithm.

BreadthFirstNodeQueue – queue for storing nodes of type *BreadthFirstNode*.

possibleEvents – array of events that form non deterministically ordered set (NOS) of events that can be possibly executed next.

nextTime – the time of the simulation if a given event is executed.

Following functions are also used:

event.node(event.attributes) – executes the function associated with particular event with parameter *event.attributes* if execution condition for the event is true.

The function schedules events and updates state variables.

executeEvent – boolean variable defined to store the result after evaluating the execution condition of the edge.

tempCalendar – copy of globalCalendar.

index – temporary variable to keep count of the elements in *possibleEvents* array.

firstNode – variable of type *BreadthFirstNode* for storing the first node on the *BreadthFirstQueue* queue.

IsSubsetOf(), *min()* and *max()* – interval math function defined in Appendix A.

The procedure ***BreadthFirstExecution*** is described using the flowchart shown in Figure 5.2.

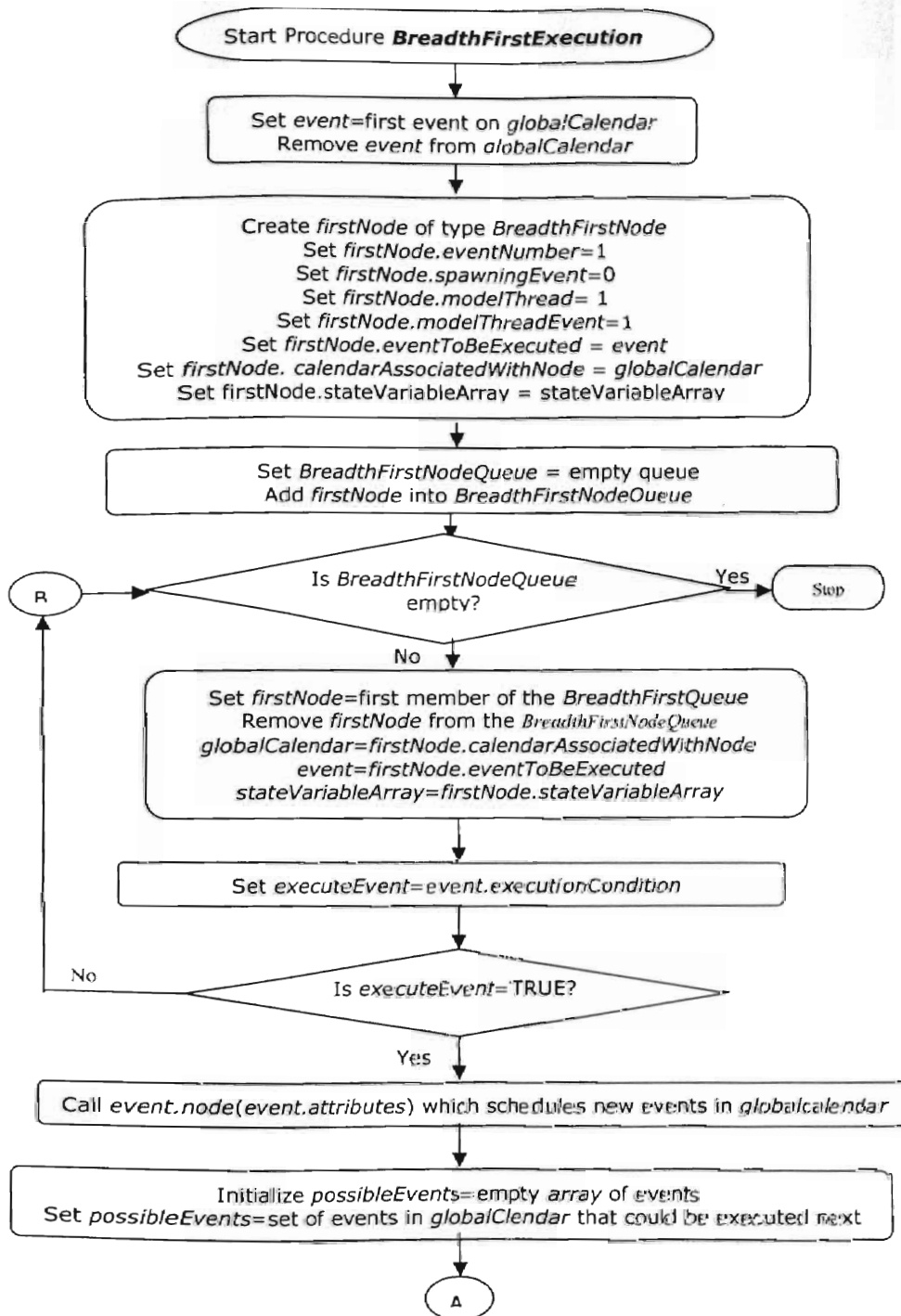


Figure 5.2: Flow chart for Breadth-First Algorithm

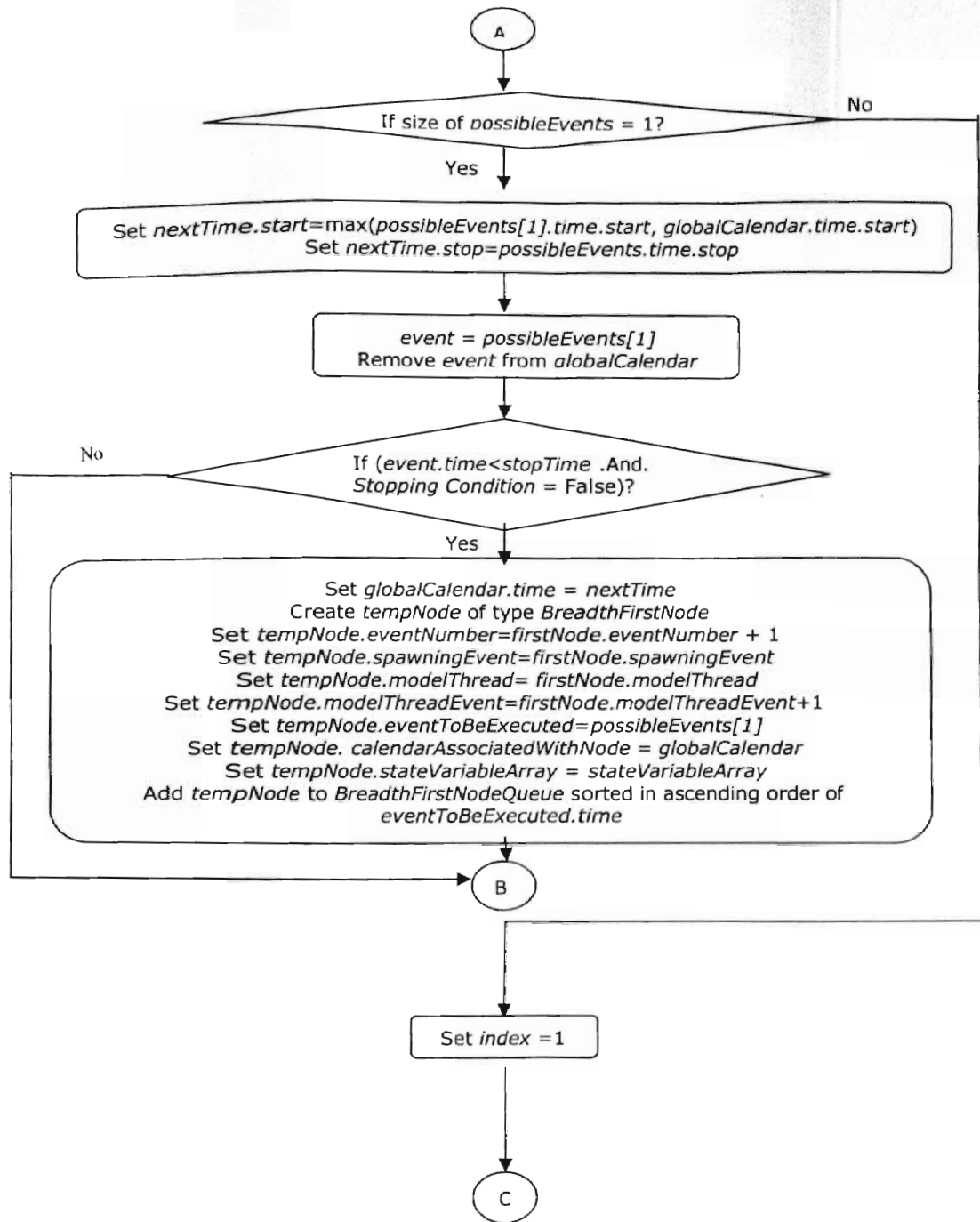


Figure 5.2: Flow chart for Breadth-First Algorithm (continued)

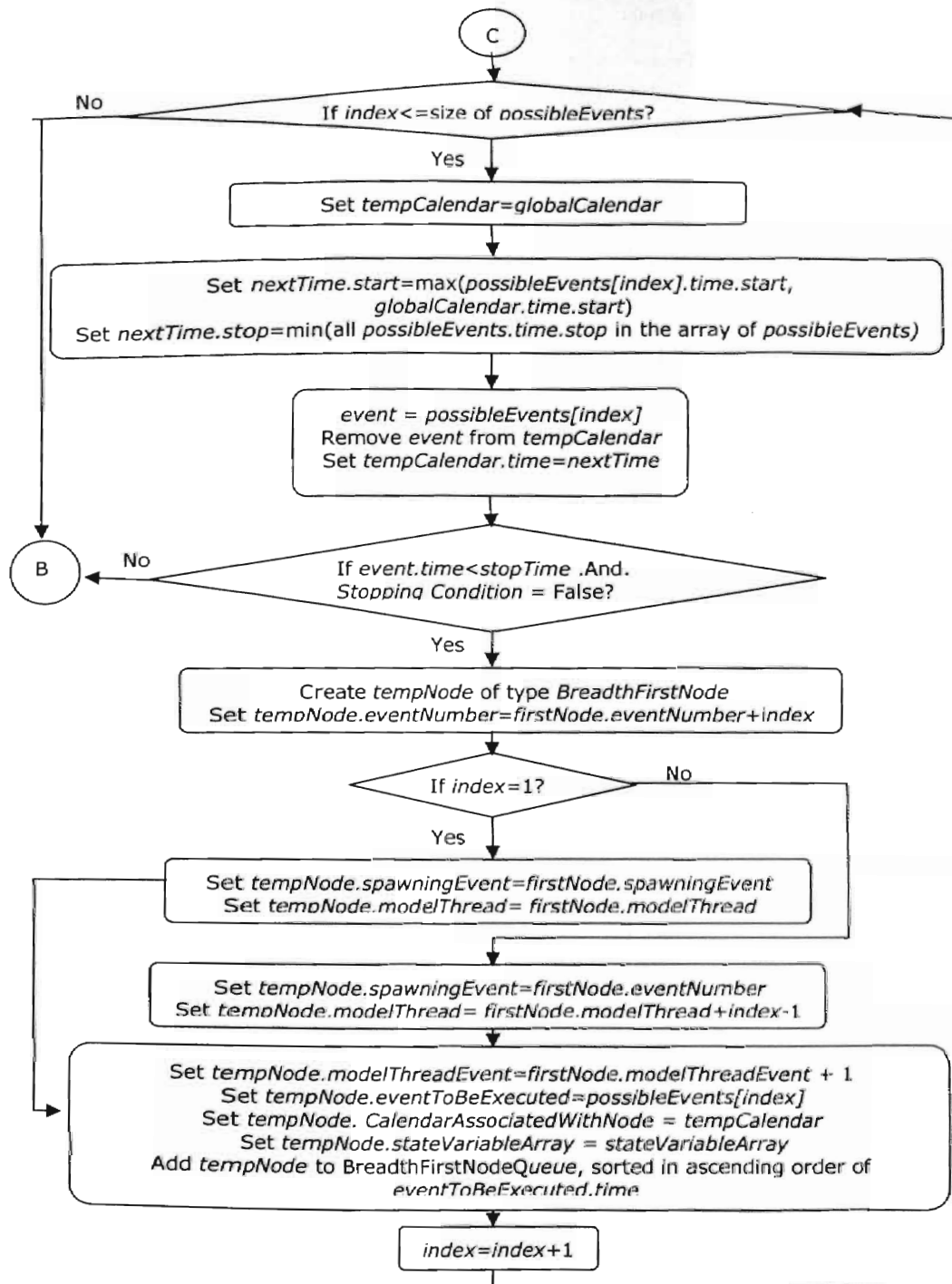


Figure 5.2: Flow chart for Breadth-First Algorithm (continued)

The the procedure *BreadthFirstExecution* algorithm is called after initialization. Two parameters are passed to this procedure and the pseudo code for the procedure are discussed below.

Procedure *BreadthFirstExecution* (first event on *globalCalendar*, *startTimeValue*)

Start

Set *event* = first event on *globalCalendar*
Remove *event* from *globalCalendar*

// Create *firstNode* using parameters *eventNumber*, *spawningEvent*, *modelThread*, *modelThreadEvent*, *eventToBeExecuted*, *globalCalendar*, *stateVariableArray*

Create *firstNode* of type *BreadthFirstNode*
Set *firstNode.eventNumber* = 1
Set *firstNode.spawningEvent* = 0
Set *firstNode.modelThread* = 1
Set *firstNode.modelThreadEvent* = 1
Set *firstNode.eventToBeExecuted* = *event*
Set *firstNode.calendarAssociatedWithNode* = *globalCalendar*
Set *firstNode.stateVariableArray* = *stateVariableArray*

//Add created node into the queue
Set *BreadthFirstNodeQueue* = empty queue for storing set of *BreadthFirstNode* sorted in ascending order of *eventToBeExecuted.time*
Add *firstNode* into *BreadthFirstNodeQueue* sorted in ascending order of *eventToBeExecuted.time*

While *BreadthFirstNodeQueue* is not empty

Remove *firstNode* from the *BreadthFirstNodeQueue*
Point *globalCalendar* to *firstNode.calendarAssociatedWithNode*
Point *event* to *firstNode.eventToBeExecuted*
Set *stateVariableArray* = *firstNode.stateVariableArray*

// Comment: Up to this point, the algorithm has removed the node from the *BreadthFirstNodeQueue* to be executed and all of the values are assigned to the global variables. This will execute the event using methods that are accessible by global variables only. The purpose of copying node values to global values is that multiple copies of the calendar have to be made while inserting new nodes into the *BreadthFirstNodeQueue*. Also the old node is to be deleted from the *BreadthFirstNodeQueue*.

Set *executeEvent* = *event.executionCondition*
If *executeEvent* = TRUE Then

// Comment: If the event can be executed, then it will call the procedure that would contain the logic to evaluate the state change i.e. changing the state of the variable and/or scheduling further events that could occur after this event. Events that are scheduled are placed in the *globalCalendar*.

Call *event.node(event.attributes)*

Else

Break out from While loop and check if there are nodes in queue.

// (i.e. go back to start of "While" loop.)

End If

// Comment: New nodes have to be added into the *BreadthFirstNodeQueue* based on events that could be executed next using this calendar. For each possible event that could be executed next a separate node has to be created and inserted in the queue. The first thing after this is to determine events that are possible and could be executed next after this event.

Initialize *possibleEvents* = empty array of events

Set *possibleEvents* = set of events in *globalCalendar* that could be executed next

// Comment: The next step is to determine the next time when the event could occur.

If (size of *possibleEvents* = 1) Then

Set *nextTime.start* = max(*possibleEvents*[1].*time.start*,
globalCalendar.time.start)

Set *nextTime.stop* = *possibleEvents.time.stop*

// Comment: If this new event occurs and the stopping condition evaluates to false then add the node into the queue with this event as a value to the variable *eventToBeExecuted*.

event = *possibleEvents*[1]

Remove *event* from *globalCalendar*

If (*event.time* < *stopTime* .AND. *Stopping Condition* = FALSE) Then

Set *globalCalendar.time* = *nextTime*

Create *tempNode* of type *BreadthFirstNode*

Set *tempNode.eventNumber* = *firstNode.eventNumber* + 1

Set *tempNode.spawningEvent* = *firstNode.spawningEvent*

Set *tempNode.modelThread* = *firstNode.modelThread*

Set *tempNode.modelThreadEvent* = *firstNode.modelThreadEvent* + 1

Set *tempNode.eventToBeExecuted* = *possibleEvents*[1]

Set *tempNode.calendarAssociatedWithNode* = *tempCalendar*

Set *tempNode.stateVariableArray* = *stateVariableArray*

Add *tempNode* to *BreadthFirstNodeQueue* sorted in ascending order of *tempNode.eventToBeExecuted.time*

End If
Else

// Comment: Insert one node each for each event that could be possible to be executed next.

```
For (index =1 to size of possibleEvents)
  Set tempCalendar = globalCalendar
  Set nextTime.start = max(possibleEvents[index].time.start,
                           globalCalendar.time.start)
  Set nextTime.stop = min(all possibleEvents.time.stop in the array of
                           possibleEvents)
  event = possibleEvents[index]
  Remove event from tempCalendar
  Set tempCalendar.time = nextTime
  If (tempCalendar.time < stopTime .AND. Stopping Condition = FALSE)
  Then
    Create tempNode of type BreadthFirstNode
    Set tempNode.eventNumber = firstNode.eventNumber + index
    If (index =1) Then
      Set tempNode.spawningEvent = firstNode.spawningEvent
      Set tempNode.modelThread = firstNode.modelThread
    Else
      Set tempNode.spawningEvent = firstNode.eventNumber
      Set tempNode.modelThread = firstNode.modelThread+index-1
    End If
    Set tempNode.modelThreadEvent = firstNode.modelThreadEvent + 1
    Set tempNode.eventToBeExecuted = possibleEvents[index]
    Set tempNode.calendarAssociatedWithNode = tempCalendar
    Set tempNode.stateVariableArray = stateVariableArray
    Add tempNode to BreadthFirstNodeQueue, sorted in ascending order
    of tempNode.eventToBeExecuted.time
  End If
End For
End If
End While Loop
End Procedure BreadthFirstExecution
```

5.4. Explanation Of The Breadth-First Algorithm With An Example

Consider the single machine example in Chapter 2. The algorithm starts its execution by calling the **Run** procedure. The *globalCalendar* is initialized with no event on this calendar. The *globalCalendar.time* is set to the time [0,0]. The stopping condition

is set to $E = 5$. The event graph for this example is shown in Figure 2.2 shows that the first event executed is the “Run” event. This node is stored in a variable that is used to store the first event to be scheduled on the calendar. Then the execution passes to the procedure *StartTime_StopTime*.

The parameter, *startTimeValue* is equal to $[0,0]$ and it indicates start time for the simulation, and *stopTimeValue* is assigned a sufficiently large value nearly equal to infinity and it indicates stop time for the simulation. These parameters are passed to the *StartTime_StopTime* procedure. The *globalCalendar.time* is set equal to *startTimeValue* = $[0,0]$. The RUN event is inserted in the global events calendar and the procedure *BreadthFirstExecution* is called with two parameters, a pointer to the RUN event on the *globalCalendar* and the starting time of the simulation, *startTimeValue*

Procedure *BreadthFirstExecution* controls the creation and execution of threads. The first event is removed from the calendar and the variable *event* is assigned the value of the first event. The next step is to create the first *BreadthFirstNode*, which is inserted in the *BreadthFirstNodeQueue*. The *BreadthFirstNode* elements will be initialized to the following values:

eventNumber = 1, *spawningEvent* = 0, *modelThread* = 1, *modelThreadEvent* = 1, *eventToBeExecuted* = *event*, *calendarAssociatedWithNode* = *globalCalendar*, and *stateVariableArray* = *stateVariableArray*. The *globalCalendar* is now because its only entry, the RUN event, has been removed and is stored in the variable *eventToBeExecuted*. The *stateVariableArray* is assigned values of state variables Q , S and E . At the start of the simulation these values are $Q = 0$, $S = 1$ and $E = 0$.

The newly created node is inserted into the *BreadthFirstNodeQueue*. Now, the *BreadthFirstNodeQueue* queue has one member, the node representing the RUN event in the first thread. Next, *firstNode* is assigned to the first member of the *BreadthFirstNodeQueue*. *firstNode* is removed from the *BreadthFirstNodeQueue* and the *globalCalendar* is assigned to the *firstNode*'s calendar (*firstNode.calendarAssociatedWithNode*). At this time, *globalCalendar* is empty because the RUN event has already been removed from the calendar. The first event notice, stored in the variable *eventToBeExecuted*, is assigned to the global variable *event*. The state variables are *Q*, *S* and *E*, and their values 0, 1 and 0, respectively are copied from *firstNode.stateVariableArray*. After all of the elements of *firstNode* are copied to local variables, the event is executed if the edge execution condition evaluates to *TRUE* by calling the edge execution function that is associated with the node. There is no scheduling edge condition on the edge between the RUN node and the ENTER node shown in Figure 2.2, so the edge execution condition is always *TRUE*. The RUN event is executed by calling the function which handles scheduling of new events and updating the state variables array. Execution of RUN event schedules an ENTER event at time $[0,0]$ and is represented as the root node of the execution tree in Figure 4.1. The event notice $([0,0], 9, ENTER, RUN-ENTER, TRUE)$ is inserted in the *globalCalendar*. A *BreadthFirstNode* node is created with the ENTER event as the event to be executed. The node is inserted in the *BreadthFirstNodeQueue*.

The process continues and executes the ENTER event and START event (nodes 2 and 3 in Figure 4.1) with the same process as the RUN event. After the START event (node 3) is executed, there are two possible events that could be executed next, an

ENTER event and a LEAVE event. The order of the execution of these two events is uncertain because the event occurrence time of these nodes overlap and they have the same priority. The ENTER event is scheduled for time [3,8] and the LEAVE event scheduled for time [4,6]. Because of this overlap, the algorithm creates two *BreadthFirstNodes*. The first node has ENTER event as the *eventToBeExecuted* and the second node has the LEAVE event as the *eventToBeExecuted*. Both nodes are inserted in the *BreadthFirstNodeQueue*. These two nodes are represented as nodes 4 and 10, respectively, in Figure 4.1.

When both *BreadthFirstNodes* are in the *BreadthFirstNodeQueue*, the first node removed from the *BreadthFirstNodeQueue* has the ENTER event as the *eventToBeExecuted*. When this event is executed, a node with the LEAVE event (node 5 in Figure 4.1) and a node with the ENTER event (node 9 in Figure 4.1) will be added to the *BreadthFirstNodeQueue*. At this point, a total of 3 nodes are in the *BreadthFirstNodeQueue*, nodes 5, 9 and 10.

The next *BreadFirstNode* to be executed will be node 10 because it has the earliest possible execution time and was in the *BreadthFirstNodeQueue* before node 5. Therefore, the execution follows the path 1, 2, 3, 4, 10, 5, 9, 11, 6, and so on. A partial execution sequence of this model is given in Table 5.2. The sequence occurs because the *BreadthFirstNodeQueue* is ordered by the execution time of the nodes.

The *BreadthFirstNodeQueue* queue is always sorted in ascending order of the event execution time. This is done to keep the simulation clock time of each thread approximately the same. The simulation clock time and state variables in each of the thread at some point of time in simulation is shown in Table 5.1.

In Table 5.1, the *BreadthFirstNodesQueue* was examined at a point in time in the simulation run. It shows that the *BreadthFirstNodeQueue* has 49 threads and that the execution times of those threads intersect with each other. Although a counter example can be shown to prove that this intersection does not always occur, it is expected that the intersection of execution times will occur often.

Table 5.1: Simulation clock time and state variables at some point in simulation.

Model Thread	Time	S	Q	E
1	[16 24]	0	1	3
2	[16 26]	0	1	3
3	[14 18]	0	2	2
4	[14 28]	0	0	3
5	[14 26]	0	0	3
6	[14 18]	0	1	2
7	[15 24]	0	1	3
8	[14 20]	0	2	2
9	[15 30]	0	0	3
10	[15 28]	0	0	3
11	[15 26]	0	0	3
12	[15 28]	0	0	3
13	[15 18]	0	2	2
14	[15 18]	0	2	2
15	[15 24]	0	1	3
16	[17 24]	0	1	3
17	[16 20]	0	2	2
18	[16 26]	0	1	3
19	[17 20]	0	0	3
20	[14 18]	0	2	2
21	[14 20]	0	1	2
22	[14 18]	0	1	2
23	[14 22]	0	1	2
24	[14 20]	0	1	2
25	[16 18]	0	2	2

Model Thread	Time	S	Q	E
26	[16 18]	0	2	2
27	[15 18]	0	2	2
28	[15 18]	0	2	2
29	[15 18]	0	2	2
30	[16 30]	0	0	3
31	[16 28]	0	0	3
32	[16 26]	0	0	3
33	[16 28]	0	0	3
34	[16 24]	0	0	3
35	[16 26]	0	0	3
36	[16 28]	0	0	3
37	[16 26]	0	0	3
38	[15 24]	0	1	3
39	[16 24]	0	1	3
40	[16 20]	0	2	2
41	[16 18]	0	2	2
42	[16 20]	0	2	2
43	[16 26]	0	1	3
44	[17 20]	0	0	3
45	[16 24]	0	1	3
46	[17 20]	0	2	2
47	[17 26]	0	1	3
48	[17 24]	0	1	3
49	[18 20]	0	0	3

Table 5.2: Partial output for single machine example using breadth-first algorithm

I – Model Thread II – Event Number III - Spawning Event
 IV- Model Thread Events V – Time VI - to Method
 S – Server Q – Queue E - Value of Exits
 X - Calendar

I	II	III	IV	V	VI	S	Q	E	X
Start Time: 07/24/2003 04:14:09									
1	1	0	1	[.00000, .00000]	begin	1	0	0	[.000,.000],enter,nil,999.,].
1	2	0	2	[.00000, .00000]	enter	1	1	0	[.000,.000],start,nil,1. [3.0000,8.00],enter,nil,999.
1	3	0	3	[.00000, .00000]	start	0	0	0	[3.00,8.00],enter,nil,999. [4.0000,6.00],leave:-1,999.
1	4	0	4	[3.0000, 6.0000]	enter	0	1	0	[4.00,6.00],leave:-1,999. [6.0000,14.0],enter,nil,999.
2	5	3	4	[4.0000, 6.0000]	leave:	1	0	1	[3.00,8.00],enter,nil,999. [4.00,6.00],start,nil,1.
1	6	0	5	[4.0000, 6.0000]	leave:	1	1	1	[6.0000,14.0],enter,nil,999. [4.00,6.00],leave:-1,999.
3	7	4	5	[6.0000, 6.0000]	enter	0	2	0	[9.0000,14.0],enter,nil,999. [7.00,16.0],enter,nil,999.
2	8	3	5	[4.0000, 8.0000]	enter	1	1	1	[4.0000,8.00],start,nil,1. [6.00,14.0],enter,nil,999.
1	9	0	6	[4.0000, 6.0000]	start	0	0	1	[8.0000,12.0],leave:-1,999. [6.00,6.00],start,nil,1.
3	10	4	6	[6.0000, 6.0000]	leave:	1	2	1	[9.0000,14.0],enter,nil,999. [7.00,16.0],enter,nil,999.
2	11	3	6	[4.0000, 8.0000]	start	0	0	1	[8.0000,14.0],leave:-1,999. [8.00,12.0],leave:-1,999.
1	12	0	7	[6.0000, 12.000]	enter	0	1	1	[9.0000,20.0],enter,nil,999.
4	13	9	7	[8.0000, 12.000]	leave:	1	0	2	[6.00,14.0],enter,nil,999. [9.00,14.0],enter,nil,999.
3	14	4	7	[6.0000, 6.0000]	start	0	1	1	[10.000,12.0],leave:-1,999.
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	561	0	17	[20.000, 30.000]	leave:	1	1	5	[19.0,38.0],enter,nil,999. [20.000,30.0],start,nil,1.
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
893	2328	2174	23	[36.000, 36.000]	leave:	1	7	5	[36.0,36.0],start,nil,1. [39.000,44.0],enter,nil,999.
894	2329	2176	23	[36.000, 36.000]	leave:	1	7	5	[36.0,36.0],start,nil,1. [39.000,44.0],enter,nil,999.
895	2330	2178	23	[36.000, 36.000]	leave:	1	7	5	[36.0,36.0],start,nil,1. [39.000,44.0],enter,nil,999.
896	2331	2179	23	[36.000, 36.000]	leave:	1	7	5	[36.0,36.0],start,nil,1. [39.000,44.0],enter,nil,999.
897	2332	2182	23	[37.000, 38.000]	leave:	1	7	5	[37.0,38.0],start,nil,1. [40.000,46.0],enter,nil,999.

5.5. Validating The Output

Using the information in Figure 4.1 and Table 5.2, it is shown that the breadth-first algorithm works as expected. The output of the breadth-first algorithm should also match the output of the depth-first algorithm. The output was obtained for the single machine system example shown in Figure 2.2 for both algorithms. Both algorithms generated 897 threads for this example, and that matched the number of threads obtained from a similar example used by Ingalls (1999). Also, the output from the breadth-first algorithm was compared event-by-event to the output of the depth-first algorithm, and all of the events were included in both outputs.

5.6. Run-Time Comparison

Literature exists that claims that depth-first traversal algorithms are faster than breadth-first traversal algorithms. In the breadth-first algorithm, since all the nodes are stored in the memory at the same time, there are large memory requirements. (Weiss, 2000) The models that were run where the same exact output is given for both the depth-first and breadth-first algorithms showed that the depth-first algorithm is always faster. In particular, for the single machine example discussed previously, the execution times on an Intel Pentium 4, 1.6 GHz for the depth-first and the breadth-first algorithms are 6 seconds and 13 seconds, respectively.

However, since the breadth-first algorithm traverses across the tree, the simulation model is able to track all active threads at any given point in time. Because of the availability of all active threads, the modeler could decide that some threads are "less valuable" than other threads. In that case, the breadth-first algorithm has the ability to

terminate these “less valuable” threads without running them to completion. The decision to terminate a thread early is based on using decision criteria defined by the modeler.

Additional coding is needed to eliminate threads from the *BreadthFirstNodeQueue*. The elimination step is added after a node is added to the queue. If the number of nodes on the queue exceeds the desired number, then the nodes on the queue are compared using the decision criteria and the node performs the worst against the decision criteria is removed from the queue. This step ensures that the number of nodes on a queue does not exceed the desired number, thereby limiting the number of active threads at any point in the simulation.

To show the speed advantage of being able to eliminate unwanted threads, the single machine server example is run with a change in the stopping condition for the simulation. In particular, the stopping condition is changed from $E = 5$ to $E = 8$, which means that an individual thread will be complete when the number of jobs processed is equal to 8. This experiment will be run for the depth-first simulation to completion. This experiment will also be run for the breadth-first simulation where the number of active threads is arbitrarily limited to 50, 100, 200, 400, and 1000. The experiment will show that, if the modeler is able to design a decision criteria that will eliminate threads, there are speed benefits to the breadth-first algorithm. Also, the output will be focused more on the areas in which the modeler is interested.

In the Table 5.3, the output from the experiment shows tremendous time savings.

Table 5.3: Run Time (Seconds) for Single Machine Example ($E = 8$)

Depth-First 99383 Threads	Breadth- First 50 Active Threads Max	Breadth- First 100 Active Threads Max	Breadth- First 200 Active Threads Max	Breadth- First 400 Active Threads Max	Breadth- First 1000 Active Threads Max
716	4	7	13	31	116

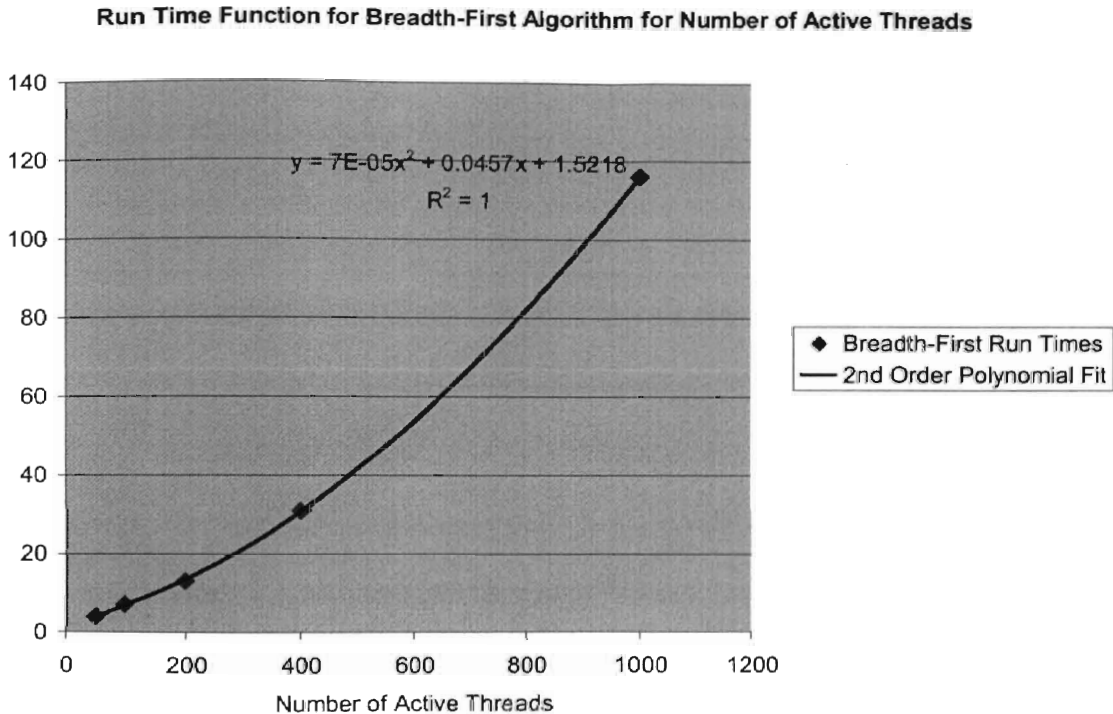


Figure 5.3: Run Time Function for Breadth-First Algorithm

Table 5.3 also shows a trend of the increase in runtime as the number of active threads increases. A chart of this run-time increase is shown in Figure 5.3. This chart also shows that a second-order polynomial perfectly fits the data in Table 5.3. With this polynomial ($0.00007x^2 + 0.0457x + 1.5218$), we can estimate the number of active threads that would have the same run-time as the depth-first algorithm that generates all

of the threads. Based on the fit, the breadth-first run-time with 2,885 active threads would be approximately the same as the depth-first run-time.

5.7. Additional Advantages Of The Breadth-First Algorithm

One of the issues with the depth-first algorithm is that model size is restricted due to the amount of memory needed to manage the stack. For single machine model, the depth-first algorithm with $E = 10$ as the stopping condition generated more than 1,000,000 threads which causes the recursion stack to overflow and stops the execution after nearly 3 hours. However, by using the breadth-first algorithm and restricting the number of nodes in the *BreadthFirstNodeQueue* to some number that would reduce the memory requirements of the model, the model could run and produce some results. The loss of information due to eliminated threads could be minimized by developing effective decision or thread elimination criteria.

Developing decision rules would be a challenge for a modeler. But it would be worth the effort because the information obtained using QS models is much more valuable when compared to traditional simulation models. This value would come from collecting information that is relevant, instead of random threads generated by the traditional simulation models. Also, eliminating unimportant threads would reduce the complexity in output analysis by allowing the modeler to concentrate on small number of important threads.

Ingalls (1999) has described the thread scoring technique for assigning scores to threads. He used thread scores to rank threads based on relative likelihood of their event execution sequence. He suggested using the thread scoring technique to eliminate threads that are less likely to occur thereby reducing the number of threads from the output of

QDES models. He also suggested that the thread explosion problem makes it difficult to get meaningful information from the output and makes output analysis difficult. The depth-first algorithm can be used for making a comparison of thread scores only after the simulation has terminated, while the breadth-first algorithm provides an opportunity to compare and eliminate low-scoring threads thereby reducing the run time of the algorithm. This thesis provides a mechanism for eliminating threads using decision criteria and thus reduces the run time for the algorithm.

Chapter 6

Summary and Future Research

6.1. Research Summary

The objective of the thesis is to develop the breadth-first algorithm for solving Qualitative Simulation Graph Models (QSGM) models. The algorithm traverses the nodes across the tree before the nodes below the tree. The algorithm enhances the depth-first algorithm developed by Ingalls (1999), by traversing the child nodes of the tree before the sibling nodes. The breadth-first algorithm developed is presented and explained in Chapter 5 using the pseudo code algorithm. QSGM, developed by Ingalls (1999), is used as modeling framework used for describing the models. The algorithm is coded in C++ using Visual C++ 6.0. The output obtained using the breadth-first algorithm is tested and compared with the output from the depth-first algorithm. The comparison of the output obtained validates the breadth-first algorithm. The parameters defined to trace the events and threads in the output shows that the breadth-first algorithm traverses nodes across the tree before going in to the depth of the solution tree. An additional model is built to test the validity of the algorithm and the output obtained from both the algorithms is compared. The output obtained from both the algorithm is exactly same (Appendix B).

The breadth-first algorithm uses a queue structure to store the sibling nodes. Each and every thread in the simulation is executed in such a way that each thread's simulation clock time is nearly equal. The reason for this is that the queue is sorted in ascending

order of simulation clock time. This may be important to the modeler in making decisions during simulation. Also, the active threads can be evaluated and compared to other threads in the simulation at any point in time, which is not possible with the depth-first algorithm. The comparison of the threads based on a certain set of decision rules at certain point in the simulation may help to eliminate threads that are unlikely to give any valuable information. One such example that uses arbitrary decision criteria for comparing and eliminating the threads is explained in Chapter 5. The example shows the importance of developing the breadth-first algorithm because it helps to improve the performance by reducing the run time using the thread elimination criteria. Also, the breadth-first algorithm reduces the complexity of the output by reducing the number of threads and allowing the modeler to concentrate threads that provide meaningful information.

The thesis accomplishes its objective by developing the breadth-first algorithm for solving qualitative simulation graph models. The algorithm provides the capability to compare all the active threads at some point in the simulation and eliminate the “less valuable” threads using decision criteria. An example is used to show that elimination of the “less valuable” threads resulted in reduction of the run time of the algorithm.

6.2. Future Research

This thesis only considered the uncertain intervals for defining qualitative time in the simulation. It did not consider constant intervals whose value must be the same throughout an entire thread in the simulation. The breadth-first algorithm can be further developed to include the logic for handling constant intervals.

This thesis does not define the state variables qualitatively. The algorithm can be developed to allow definition of the state variables qualitatively. This may increase the number of threads and the run time of the algorithm, but it would provide flexibility in defining of the modeling parameters.

One of the major concerns of the qualitative simulation algorithms is execution time. Threads can explode exponentially and the execution time for the qualitative simulation can be very long. It is clear that one of the benefits of breadth-first simulation is the use of decision criteria to eliminate active threads. In future research, effective decision criteria can be designed which can be used to compare and evaluate the threads in qualitative simulation. With effective decision criteria, unimportant threads can be terminated thereby reducing the run time for the algorithm. Thread scoring techniques, such as those suggested by Ingalls (1999), can also be developed to eliminate the unimportant threads. The breadth-first algorithm provides the groundwork for such type of future research by providing a tool to solve QDES models. Algorithms that use parallel processors and multithreading can be developed to execute QSGM models to reduce the run time of the models.

The QSGM methodology developed has not received much attention in practical applications. This thesis attempts to provide an alternative tool for solving QSGM models which might allow QSGM to be developed for practical applications.

References

- Banks, J., 1998. *Handbook of Simulation*, Engineering and Management Press.
- Berleant, D., and Kuipers, B. J., 1997. Qualitative and Quantitative Simulation: Bridging The Gap, *Artificial Intelligence*, 95:215-255.
- Bulitko, V., and Wilkins, D. C., 2003. Qualitative Simulation of Temporal Concurrent Processes using Time Interval Petri Nets, *Artificial Intelligence*, 144:95-124.
- Cellier, F. E., 1991. Qualitative Modeling And Simulation: Promise Or Illusion, *Proceedings of the 1991 Winter Simulation Conference*, eds. Barry L. Nelson, W. David Kelton, and Gordon M. Clark, pp. 1086-1090.
- Clancy, D. J., Brajnik, G., and Kay, H., 1997. Model Revision: Techniques and Tools For Analyzing Simulation Results and Revising Qualitative Models, *11th International Workshop on Qualitative Reasoning*, Cortona, Siena Italy, June 1997.
- Damerdjji, H. And Nakayama, M. K., 1999. Two-Stage Multiple-Comparison Procedures For Steady-State Simulations, *ACM Transactions on Modeling and Computer Simulation*, 9(1):1-30.
- Farquhar, A., Kuipers, B., Rickel, J., Throop, D. and The Qualitative Reasoning Group, 1994. *QSIM: The Program and its Use*, Department of Computer Science, University of Texas at Austin, Austin, Texas, October 18, 1994. [Retrieved on July 29, 2003] URL: <http://www.cs.utexas.edu/users/qr/QR-software.html> - qsim.
- Goldsman, D., Marshall, W. S., Seong-Hee Kim, and Nelson, B. L., 2000. "Ranking And Selection For Steady-State Simulation", *Proceedings of the 2000 Winter Simulation Conference*, eds. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, pp. 544-553.
- Hamscher, W., Kiyang, M. Y. and Lang, R., 1995. Qualitative Reasoning in Business, Finance, and Economics: Introduction, *Decision Support Systems*, 15(2):99-103.
- Hao, W., Fujimoto, R. M. and Riley, G., 2001. Experiences Parallelizing A Commercial Network Simulator, *Proceedings of the 2001 Winter Simulation Conference*, eds. B.A. Peters, J.S. Smith, D.J. Mederios, and M.W. Rohrer, pp. 1353-1360.
- Hocaoglu, M.F., 2003. A Comparison between Qualitative Simulation and Traditional Simulation: Bridging the Conceptual Gap, Applied Modeling and Simulation Joint Research Group (AMSJRG), Information Technologies Research Institute Tubitak-Marmara Research Center, Turkey. Working Paper. [Retrieved on July 29, 2003] URL: <http://www.btae.mam.gov.tr/~hocaoglu/qscomparison.pdf>.

- Ingalls, R. G., 1999. *Qualitative Simulation Graph Methodology and Implementation*, The University of Texas at Austin.
- Kuipers, B., 1986. Qualitative Simulation, *Artificial Intelligence*, 29:289-238.
- Kuipers, B., 2001. *Qualitative Simulation*, 2001. [Retrieved on July 22, 2003] URL: <http://www.cs.utexas.edu/users/qr/papers/Kuipers-epst-01.html>.
- McCormack, W. M., and Sargent, R. G., 1981. Analysis of future Event Set Algorithms for Discrete-event Simulation, *Communications of the ACM*, 24(12):801-812.
- Neelamkavil, F., 1987. *Computer Simulation and Modelling*, John Wiley & Sons Ltd.
- Preiss, B. R., 1999. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*, John Wiley & Sons, Inc. New York.
- Schruben, L.W., 1983. Simulation Modeling with Event Graphs, *Communications of the ACM*, 26(11):957-963.
- Weiss, M. A., 2000. *Data Structures And Problem Solving Using C++*, Addison-Wesley. Reading, Massachusetts.
- Wyatt, G. J., Leitch, R. R. and Steele, A.D., 1995. Qualitative and Quantitative Simulation of Interacting Markets, *Decision Support Systems*, 15(2):105-113.
- Yücesan E. and L. W. Schruben. 1992. Structural and Behavioral Equivalence of Simulation Models. *ACM Transactions on Modeling and Computer Simulation*, 2(1): 82-103.

Appendix A. Interval Math

Consider two interval $a=[a^-,a^+]$ and $b=[b^-,b^+]$.

Function	Result of the function
$a=b$	Assign ($a^- = b^-$) and ($a^+ = b^+$)
$a+b$	Evaluates to $[a^-+b^-, a^++b^+]$
$a-b$	Evaluates to $[a^- - b^+, a^+ - b^-]$
$a*b$	Evaluates to $[a^- * b^-, a^+ * b^+]$
$a \text{ IsSubsetOf } b$	True if $((a^- > b^-) \text{ and } (a^+ < b^+))$ else False
$\max(a,b)$	Evaluates to $[\max(a^-,b^-), \max(a^+,b^+)]$
$\min(a,b)$	Evaluates to $[\min(a^-,b^-), \min(a^+,b^+)]$
$a < b$	True if $(a^+ < b^-)$ else False
$a \text{ equal to } b$	True if $((a^- = b^-) \text{ and } (a^+ = b^+))$ else False
$\text{Midpoint}(a)$	Equals to $((a^- + a^+)/2)$

Source: Ingalls (1999) Page 25.

Appendix B. PERT Network Example

The breadth-first algorithm is also validated using the PERT (Program Evaluation and Review Technique) network problem in addition to the single machine server example discussed in the thesis. The PERT example discussed here is a simple network with 7 nodes. The event graph representation of the PERT network is shown in Figure B.1.

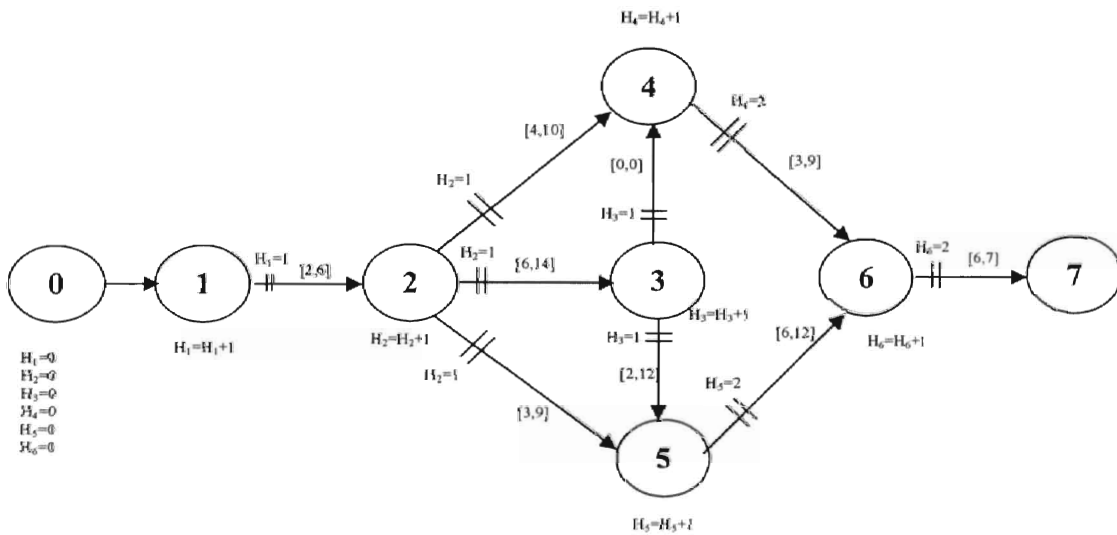


Figure B.1. PERT Network Event Graph

The nodes are labeled from 0 to 7 and the scheduling edge conditions are represented on the edges. The state variables are defined as H_i , for $i = 1$ to 6.

H_i - the number of times node i has been hit. For example, when the edge between nodes 1 and 2 is scheduled, the number of hits on node 2 will be equal to 1.

The partial output obtained using the depth-first algorithm is shown in table B.1.

Table B.1: The Depth-First Algorithm Partial Output for PERT Network Example

I – Model Thread

II – Event Number

III - Spawning Event

IV- Model Thread Events

V – Time

VI – Head Event

VII – Tail Event

VIII - Calendar

I	II	III	IV	V	VI	VII	VIII
1	1	0	1	[.000,.000]	0	1	[2.0000,6.0000],node:,(1 2),3,nil.
1	2	0	2	[2.00,6.00]	1	2	[8.0000,20.000],node:,(2 3),3,nil. [6.0000,16.000],node:,(2 4),3,nil. [5.0000,15.000],node:,(2 5),3,nil.
1	3	0	3	[8.00,15.0]	2	3	[6.0000,16.000],node:,(2 4),3,nil. [5.0000,15.000],node:,(2 5),3,nil. [8.0000,15.000],node:,(3 4),2,nil. [10.000,27.000],node:,(3 5),3,nil.
1	4	0	4	[8.00,15.0]	3	4	[6.0000,16.000],node:,(2 4),3,nil. [5.0000,15.000],node:,(2 5),3,nil. [10.000,27.000],node:,(3 5),3,nil.
1	5	0	5	[8.00,15.0]	2	4	[5.0000,15.000],node:,(2 5),3,nil. [10.000,27.000],node:,(3 5),3,nil. [11.000,24.000],node:,(4 6),3,nil.
1	6	0	6	[8.00,15.0]	2	5	[10.000,27.000],node:,(3 5),3,nil. [11.000,24.000],node:,(4 6),3,nil.
1	7	0	7	[10.0,24.0]	3	5	[11.000,24.000],node:,(4 6),3,nil. [16.000,36.000],node:,(5 6),3,nil.
1	8	0	8	[11.0,24.0]	4	6	[16.000,36.000],node:,(5 6),3,nil.
1	9	0	9	[16.0,36.0]	5	6	[22.000,43.000],node:,(6 7),3,nil.
1	10	0	10	[22.0,43.0]	6	7	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
39	173	169	8	[16.0,29.0]	5	6	[11.000,29.000],node:,(4 6),3,nil.
39	174	169	9	[16.0,29.0]	4	6	[22.000,36.000],node:,(6 7),3,nil.
39	175	169	10	[22.0,36.0]	6	7	
40	176	168	7	[11.0,29.0]	4	6	[10.000,32.000],node:,(3 5),3,nil.
40	177	168	8	[11.0,32.0]	3	5	[17.000,44.000],node:,(5 6),3,nil.
40	178	168	9	[17.0,44.0]	5	6	[23.000,51.000],node:,(6 7),3,nil.
40	179	168	10	[23.0,51.0]	6	7	

The output shows threads 1, 39 and 40 only. The total number of threads for the PERT example represented by event graph in Figure B.1 produces 40 threads. The above model is also executed using the breadth-first algorithm and the output is shown in Table B.2. It should be noted that the events in Table B.2 are not in the order that they were executed. The order of execution is shown with the *Event Number* column.

Table B.2: The Breadth-First Algorithm Partial Output for PERT Network Example

I – Model Thread II – Event Number III – Spawning Event
IV – Model Thread Events V – Time VI – Head Event
VII – Tail Event VIII – Calendar

I	II	III	IV	V	VI	VII	VIII
1	1	0	1	[.000,.000]	0	1	[2.0000,6.0000],node:,(1 2),3,nil
1	2	0	2	[2.00,6.00]	1	2	[8.0000,20.000],node:,(2 3),3,nil. [6.0000,16.000],node:,(2 4),3,nil. [5.0000,15.000],node:,(2 5),3,nil.
1	3	0	3	[8.00,15.0]	2	3	[6.0000,16.000],node:,(2 4),3,nil. [5.0000,15.000],node:,(2 5),3,nil. [8.0000,15.000],node:,(3 4),2,nil. [10.000,27.000],node:,(3 5),3,nil.
1	6	0	4	[8.00,15.0]	3	4	[6.0000,16.000],node:,(2 4),3nil. [5.0000,15.000],node:,(2 5),3,nil. [10.000,27.000],node:,(3 5),3,nil.
1	11	0	5	[8.00,15.0]	2	4	[5.0000,15.000],node:,(2 5),3,nil. [10.000,27.000],node:,(3 5),3,nil. [11.000,24.000],node:,(4 6),3,nil.
1	18	0	6	[8.00,15.0]	2	5	[10.000,27.000],node:,(3 5),3,nil. [11.000,24.000],node:,(4 6),3,nil.
1	32	0	7	[10.0,24.0]	3	5	[11.000,24.000],node:,(4 6),3,nil. [16.000,36.000],node:,(5 6),3,nil.
1	60	0	8	[11.0,24.0]	4	6	[16.000,36.000],node:,(5 6),3,nil.
1	100	0	9	[16.0,36.0]	5	6	[22.000,43.000],node:,(6 7),3,nil.
1	140	0	10	[22.0,43.0]	6	7	
25	53	28	7	[11.0,29.0]	4	6	[10.000,32.000],node:,(3 5),3,nil.
25	90	28	8	[11.0,32.0]	3	5	[17.000,44.000],node:,(5 6),3,nil.
25	130	28	9	[17.0,44.0]	5	6	[23.000,51.000],node:,(6 7),3,nil.
25	170	28	10	[23.0,51.0]	6	7	

40	98	58	8	[16.0,29.0]	5	6	[11.000,29.000],node:,(4 6),3,nil.
40	138	58	9	[16.0,29.0]	4	6	[22.000,36.000],node:,(6 7),3,nil.
40	178	58	10	[22.0,36.0]	6	7	

Both the algorithms produce similar output which can be seen from Table B.1 and Table B.2. Threads 1, 39 and 40 of the depth-first algorithm are exactly same as threads 1, 40 and 35 of the breadth-first algorithm. This further validates the accuracy of the breadth-first algorithm.

VITA

②

Nitin Agrawal

Candidate for the Degree of

Master of Science

Thesis: BREADTH-FIRST ALGORITHM FOR QUALITATIVE DISCRETE-EVENT
SIMULATION.

Major Field: Industrial Engineering and Management.

Biographical:

Personal Data: Born in Amravati, Maharashtra, India on April 8, 1977, son of
Satyanarayan Agrawal and Usha Agrawal.

Education: Graduated from Brijlala Biyani Science College, Amravati,
Maharashtra, India in May 1994; received Bachelor of Engineering
degree in Production from V. J. T. I., University of Mumbai, Mumbai
in June 1998. Completed the requirements for the Master of Science
degree with a major in Industrial Engineering and Management at
Oklahoma State University in December 2003.

Experience: Employed as a graduate Teaching Assistant, School of Industrial
Engineering and Management, Oklahoma State University, August
2003 to December 2003. Employed as a graduate Research Assistant,
Oklahoma State University, School of Industrial Engineering and
Management, January 2002 to May 2003. Employed as an Assistant
Manager, Engine Assembly 3-Wheeler Division, Bajaj Auto Ltd.,
India, July 1998 to July 2001.

Professional Membership: Alpha Pi Mu and Kappa Phi Kappa.