The Implementation of Temporal Intervals in Qualitative Simulation Graphs

RICKI G. INGALLS
Compaq Computer Corporation
and
DOUGLAS J. MORRICE and ANDREW B. WHINSTON
The University of Texas at Austin

In this paper we develop and implement a simulation modeling methodology that combines discrete event simulation with qualitative simulation. Our main reason for doing so is to extend the application of discrete event simulation to systems found in business for which precise quantitative information is lacking. The approach discussed in this paper is the implementation of temporal interval specifications in the discrete event model and the construction of a temporal interval clock for the qualitative simulation model.

Categories and Subject Descriptors: I.6.1 [**Simulation and Modeling**]: Simulation Theory—*Model classification*; I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Combined*; *Discrete event*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Discrete event simulation, event graphs, qualitative simulation, simulation modeling

1. INTRODUCTION

In this paper we develop and implement a simulation modeling methodology called qualitative discrete event simulation (QDES) that combines discrete event simulation (DES) with qualitative simulation. Qualitative simulation was originally developed in the physical sciences [Forbus 1988], but more recently has found application in economics and business [Hinkkanen et al. 1993]. The qualitative approach is useful when the level of

Authors' addresses: R. G. Ingalls, Supply Chain Modeling Group, Global Integrated Logistics, Compaq Computer Corporation, 20555 SH 249, Houston, TX 77070; email: ricki.ingalls@compaq.com; D. J. Morrice and A. B. Whinston, Department of Management Science and Information Systems, Graduate School of Business, The University of Texas at Austin, Austin, TX 78712-1175; email: morrice@mail.utexas.edu; abw@uts.cc.utexas.edu.

knowledge about the system being modeled is imprecise. In fact, qualitative simulation is designed to represent whatever level of knowledge is available. For example, variables describing the state of a system might be represented in a qualitative simulation model as simply increasing, decreasing, or constant with respect to time if no other information is available. Inferences derived from the results of a qualitative simulation model, although less precise, are often considered more general and robust, since these inferences do not rely on precise and perhaps faulty assumptions.

DES suffers from many of the same problems that motivated the pioneers of qualitative simulation. The amount of detail needed in a simulation is often overwhelming. Statistical distributions used in the simulations are often based on incomplete information or the intuition of the model builder. Because of the very nature of DES, appropriate levels of abstraction are often difficult to determine and justify. These problems make DES models difficult to build and verify. The simulation expert can usually discredit a simulation by criticizing the input distributions, the appropriate level of detail, etc. Decision makers often consider simulation impractical because of costly data collection requirements, long and costly model development cycles, lengthy analysis times, and the difficulty of evaluating alternatives.

Our main reason for developing QDES is to extend the application of DES to systems found in business for which precise quantitative information is lacking. DES can be qualitatively defined by permitting imprecise specification of elements that are typically quantitatively specified either deterministically or in the form of a probability distribution. In this paper we assume imprecise specification of event occurences. More specifically, closed intervals in $\Re$ (the set of real numbers) are used to represent event execution time uncertainty. The intervals are called temporal intervals. We implement QDES using temporal intervals and an interval event calendar. As a result of our focus on imprecise specification of event occurences in this paper, only time-based variables such as interarrival times and service times are qualitatively specified. Ingalls [1999] extends QDES to include qualitative specification of other state variables that are not based on the simulation time clock (for example, order size or crew size). In instances where it is important to distinguish between QDES and regular DES, we use RDES to represent the latter.

Our approach is also motivated by a classification of problems in decision making: deterministic, risky, and uncertain, proposed by Luce and Raiffa [1954]. The deterministic classification includes problems for which all information required for a solution is completely known. The risky classification includes all problems for which at least some of the information required for a solution is specified probabilistically (i.e., in the form of probability distributions). In uncertain problems, at least some of the information required for a solution cannot be specified even in the form of probability distributions. Kouvelis and Yu [1997] apply this classification to optimization problems, and distinguish between deterministic mathematical programs, stochastic programming (for risky problems) and their work in robust optimization (for uncertain problems).

If the above classification were used for DES, the RDES and QDES approaches would be considered risky and uncertain, respectively. Therefore, QDES could also be thought of as an approach to robust discrete event simulation. QDES is robust in two senses. First, it requires fewer modeling assumptions than regular discrete event simulation. In particular, it does not require the specification of probability distributions for the timing of events. Therefore, erroneous assumptions and inferences are less likely.

The second sense in which QDES is robust has to do with what the simulation generates. RDES relies on sampling from input probability distributions to generate output probability distributions. In contrast, QDES generates event sequences or *processes* [Nance 1981] and the corresponding uncertainty in the event timings in the form of temporal intervals. When two or more event execution intervals intersect during the execution of a QDES simulation, the event execution order becomes uncertain and multiple processes result. QDES simulates and generates all possible processes and the corresponding uncertainty in their event timings. In this sense, QDES characterizes all possible outcomes of the simulation. The characterization of all possible outcomes is called *coverage*. Coverage is an important advantage of the QDES approach because it does not miss outcomes that a sampling-based approach such as RDES might with a finite sample size. However, coverage does not come without a price. Process explosion can occur making it computationally impractical to use the QDES approach. In light of these considerations, we view QDES as useful for a different class of problems than generally considered by RDES. Whereas typically RDES is used to model and analyze operations problems, QDES should be used to model and analyze tactical or strategic problems where information is usually more qualitative (due to aggregation or lack of availability) and event granularity is higher.

We develop our methodology within the modeling framework of event graphs and simulation graph models [Schruben 1983; Som and Sargent 1989; Yücesan 1989; Yücesan and Schruben 1992; Schruben and Yücesan 1993; Ingalls et al. 1996]. Simulation graph models are used because they provide a simple yet general representation of a discrete event simulation [Yücesan 1989]. The concepts in this paper were introduced in Ingalls et al. [1994].

The remainder of the paper is organized in the following manner. Section 2 contains a background discussion on qualitative simulation, DES, and event graphs. Section 3 describes interval value modeling and interval arithmetic. Additionally, it contains a description of the types of temporal intervals used in QDES. In Section 4, we develop the QDES methodology and describe it in the form of detailed algorithms. Section 5 demonstrates the QDES approach on an example of simple inbound/outbound logistics pipeline model. Section 6 contains our conclusions and a discussion of future research.

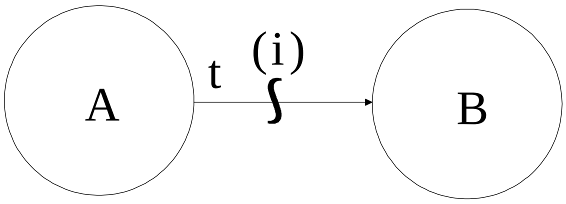

Fig. 1. Scheduling edge.

2. BACKGROUND

2.1 Qualitative Simulation

Qualitative simulation was developed to describe complex physical phenomena in the absence of good quantitative information [Forbus 1988]. The qualitative simulation uses qualitative state variables. There are several implementations of qualitative state variables. Interval valued variables can be implemented as continuous intervals on $\Re$. Ordinal valued variables are completely enumerated and ranked. Signed variables, depending on the implementation, can have one of the values in the set *[-,+]* or the set *[-,0,+]*. Sign variables are a type of ordinal valued variable, which is a subset of an interval valued variable. An example of an ordinal set is to mark time landmarks [Kuipers 1987] as distinct time points.

We assume that a closed time interval in $\Re$, called a temporal interval, can be specified for each event that contains the actual occurrence with certainty. The level of uncertainty about the exact timing of the event occurrence is reflected in the width of the interval. The more uncertainty there is in an event timing, the wider the corresponding temporal interval.

2.2 Event Graphs and Simulation Graph Models

Schruben [1983] introduced event graphs (EGs) as a modeling paradigm for DES. The basis for an event graph model is represented in Figure 1. The vertices labeled A and B represent events. The edge specifies that there is a relationship between the two events. More specifically, the construct, called a scheduling edge, can be interpreted as follows: "whenever event A occurs, if condition *(i)* is true then event B will be scheduled to occur t time units later" [Schruben 1983]. The quantity t may assume the value zero, in which case B happens at the same instant as A. Note that it is possible (and often necessary) to specify an edge with no condition.

Figure 2 depicts a second basic construct in EGs, referred to as a *canceling edge*. The canceling edge has the following interpretation: "if condition *(i)* is true at the instant event A occurs, then the currently scheduled event B will be canceled t time units later" [Schruben 1983]. In most instances, t equals zero for the canceling edge [Schruben 1995, p. 76]. The canceling edge is used to delete scheduled transactions that execute the vertex that is at the head of the edge. The canceling edge was added to

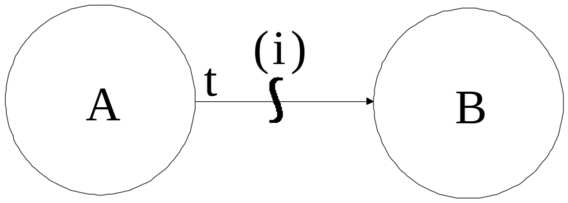

Fig. 2. Canceling edge.

the event graph framework to handle disruptions such as machine breakdowns in a manufacturing simulation.

It is also possible to parameterize the event vertices in the event graph, and thus extend the basic constructs in Figures 1 and 2. Parameterization is accomplished through vertex parameters and edge attributes. A vertex parameter list is a string of state variables associated with a particular vertex. An edge attribute list is a string of expressions associated with a particular edge. These lists are used in scheduling or canceling specific instances of events. Using these parameters is analogous to passing values in subroutines in high-level programming languages. For example, Figure 3 provides an extension of Figure 1. The construct in Figure 3 is interpreted as follows: "if condition (i) is true at the instant event A occurs, then event B(j) will be scheduled to occur *t* time units later with parameter string *j*, equal to *k*" [Schruben 1995, p. 79].

Simulation graph models (SGMs) [Yücesan 1989; Yücesan and Schruben 1992; Schruben and Yücesan 1993] provide a formalization of EGs using graph theory. In fact, they are designed to provide a formalism for DES models in general. The SGM framework is useful for establishing theoretical properties such as the generality of EG models [Yücesan 1989; Chap. 4]. Additionally, this framework greatly facilitates the development of simulation execution algorithms. We use the SGM framework in Section 4 to develop the QDES execution algorithm.

Ingalls et al. [1996] extends EGs using SGMs by eliminating the canceling edge and replacing it with the *edge execution condition*. The basic construct of the EG with the edge execution condition is given in Figure 4. The nodes labeled A and B represent events, and the edge specifies that there is a relationship between the two events. The construct is interpreted as follows: "if condition (i) is true at the instant event A occurs, then event B will be scheduled to occur *t* time units later. Event B will be executed *t* time units later with the state variables in array n set equal to the values in array k if condition (j) is true *t* time units later." We use the EG construct in Figure 4 to develop and implement QDES.

3. INTERVAL VALUES IN MODELING

3.1 Overview

Interval values on $\Re$ have been used in the qualitative analysis of continuous systems by Kiang et al. [1995] and Ingalls et al. [1996]. The purpose

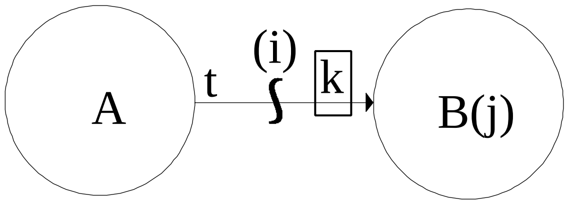

Fig. 3. Vertex parameters and edge attributes on a scheduling edge.

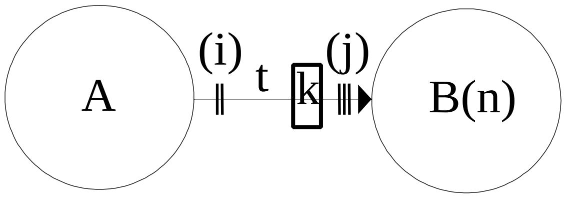

Fig. 4. Event graph with execution conditions.

for describing state variables with interval values is to allow the user to describe the inherent uncertainty of the decision maker or modeler when it comes to the true value of the variable. Interval specification allows the decision maker to refine the estimate of the true variable value as more information is gathered. For example, if an input to the model is demand, and the (unknown) actual demand value is 42, the decision maker may be able to run experiments or by experience determine that the true demand lies in the range [35,45]. In this paper, we only allow the interval value specification to be used to describe activity times (or delay times) in the discrete event model.

3.2 Interval Math

In order to be able to handle interval values, we adopt the conventions of Allen [1983]. Let $s = [s^-, s^+]$ and $t = [t^-, t^+]$ be closed intervals in $\Re$. Figure 5 gives the algebra that is used for intervals in this QDES implementation. We have extended this algebra to handle open and half-open intervals. We have also implemented logical extensions such as “and” (&), “or” (|), and “negation” (!) for use in boolean expressions.

3.3 Constant Intervals

In order to handle the use of intervals in the methodology better, we must create some definitions about the type (or use) of the intervals in the model. The first is a *constant interval*. A constant interval is an interval whose value must be the same throughout the entire process of the simulation, i.e., it is assumed that the actual value of the variable is a constant that lies somewhere in an interval. These types of intervals are best used in design decisions, in which a modeler can plan for, or even force, the exact same activity time every time that a particular activity takes place. In the model, the user must put in an initial condition for a constant interval.

Relation	Symbol	Symbol for Inverse	Definition
t before s	<	>	$t^+ < s^-$
t equals s	=		$(t^- = s^-)$ and $(t^+ = s^+)$
t overlaps s	o	oi	$(t^- < s^-)$ and $(t^+ > s^-)$ and $(t^+ < s^+)$
t meets s	m	mi	$t^+ = s^-$
t during s	d	di	$((t^- > s^-)$ and $(t^+ <= s^+))$or $((t^- >= s^-)$ and $(t^+ < s^+))$
t starts s	s	si	$t^- = s^-$
t finishes s	f	fi	$t^+ = s^+$
In addition, we define the following:			
t intersects s	i		(t overlaps s) | (s overlaps t) | (t during s) | (s during t)
t + s	+		$[t^- + s^-, t^+ + s^+]$
t – s	-		$[t^- - s^-, t^+ - s^+]$
max(t,s)	max		$[\max(t^-,s^-),\max(t^+,s^+)]$
min(t,s)	min		$[\min(t^-,s^-),\min(t^+,s^+)]$
intersection(t,s)	$\cap$		$[\max(t^-,s^-),\min(t^+,s^+)]$,if $(\max(t^-,s^-) \leq \min(t^+,s^+))$, $\varnothing$,otherwise

Fig. 5. Interval algebra for intervals $s = [s^-, s^+]$ and $t = [t^-, t^+]$.

This initial condition constitutes the lower and upper bounds of where the constant can lie. The simulation then takes that interval and changes it so that the resulting interval is logically consistent throughout the entire process. From an event graph standpoint, this should primarily be used in graphs that have cycles and where the modeler wants a constant to be used somewhere in the cycle. Section 5.2 provides an example application of constant intervals.

3.4 Uncertain Intervals

The second type of interval is the *uncertain interval*. This is an interval whose value could be different every time that the interval is evaluated. It is the modeling equivalent of sampling from an unknown probability distribution that is bounded by the interval that is input by the user. This is certainly the most common interval used in a simulation. For noncyclical networks, such as PERT networks and job shop scheduling problems, this is the only type of interval that should be used.

4. TEMPORAL INTERVALS AND QUALITATIVE TIME EVENT CALENDAR CONSTRUCTS

A *temporal interval* allows the user to define time as an interval in the simulation. It can be defined as either a constant interval or an uncertain interval. A special temporal interval in the simulation is the current simulation time. Having a temporal interval for the simulation time means that current state of the simulation can occur at any time during the interval defined by the current simulation time τ, where τ is an interval $[\tau^-, \tau^+]$. The definition of a temporal interval for the system simulation time leads to changes in the event calendar and the event notices on the event calendar.

An *event notice* is a record that includes the time that the event is scheduled to be executed, the vertex that scheduled the event, the time that the event was scheduled, the vertex that will be executed, priority of the event, the execution condition, and any attributes that are passed to the vertex that will be executed. It is important to note that the time the event is scheduled to be executed is a temporal interval. Assuming that l is an event notice, then t_l is the designation for the time at which event notice v_l can occur. b_l denotes the time that the event notice was scheduled. v_l denotes the vertex that will be executed when the event notice is taken off of the calendar. γ_l denotes the priority of the event notice with lower values denoting a higher priority. a_l is an array of attribute values for the event notice.

An *event calendar* is an ordered collection of event notices that are ordered where each event notice's time is ascending. Specifically, if a and b are two event notices, for a to precede b on the event calendar, one of the following conditions must apply:

$$t_a^- < t_b^-$$

$$t_a^- = t_b^- \text{ and } t_a^+ = t_b^+$$

Notice that this ordering of the calendar guarantees that the first event notice on the event calendar is the candidate to be executed next without regard to priority. Also, since each process has its own event calendar, the event calendar has an attribute τ that tracks the current time of the simulation for that process.

4.1 Execution Constructs

An event calendar is a method for determining the order of events in DES. An RDES event calendar is a strongly ordered list of event notices. Typically, the event calendar is sorted by time where the earliest events are at the front of the list. If there are event notices that are scheduled to occur at the same time, the order is determined by some predetermined rule or a user priority. Predetermined rules vary from first-on, first-off to random, and vary according to the implementor's discretion. Regardless, the event calendar maintains its strongly ordered characteristics.

In a QDES model with constant delay times, it is likely that there will be ties on the event calendar. But unlike quantitative RDES, the qualitative implementation would not assume a tie breaking strategy. Rather, the qualitative model would create a process for every possible ordering of the ties. Thus, the qualitative event calendar loses the strongly ordered characteristic of its quantitative counterpart. In fact, the qualitative event calendar becomes the union of many nondeterministically ordered sets (NOSs). In essence, an NOS is a set of event notices on the event calendar whose execution order is uncertain. We call the members of an NOS nondeterministically ordered events (NOEs). Because we do not have a strongly ordered event calendar and because one of the foundations of qualitative simulation is coverage, the QDES creates different processes to execute all the possible orderings of an NOS.

When the order of event notices is uncertain, the simulation generates a separate process for each event notice, with that event notice being executed first. For any two event notices a and b, the sequence of the event notice is considered uncertain if $t_a \cap t_b$ and $\gamma_a \gamma_b$. When these conditions exist, then the set of event notices is an NOS. In the qualitative model, the event calendar is made up of a union of NOSs. For the event calendar mechanism to work correctly, we must be able to determine the NOS that will be executed next, given the current state of the event calendar. The implementation of the event calendar for NOS in QDES is called the temporal interval calendar *(TIC)*.

Upon determining the nondeterministically ordered event notices that make up the NOS, the qualitative simulation creates a process for each event in the set. In process i, the ith event notice in the set becomes the first event notice to be executed. The remaining event notices in the set remain in the event calendar for that process. Each process maintains its own event calendar and calendar time. Next, we rigorously define and develop QDES using the simulation graph framework.

4.2 Design and Development of QDES Using Simulation Graphs

The simulation graph framework introduced by Yücesan and Schruben [1992]; Schruben and Yücesan [1993], and extended by Ingalls et al. [1996] provides a general framework in which to define and develop QDES. Let $G = (V(G), E(G), \psi_G)$ be a directed graph where $V(G)$ is the set of vertices, $E(G)$ is the set of edges, and ψ_G is an incidence function that associates with each edge of G an ordered pair of (not necessarily distinct) vertices of G [Bondy and Murty 1976]. G can be a multiarc graph, meaning that more than one edge can connect the same two vertices directed in the same direction. Vertices and edges can have functions and attributes assigned to them, making the model a network, since a network is regarded as a graph in which additional data are stored at the vertices and edges [Lawler 1976].

The function that is assigned to the vertices is $F = \{f_v : v \epsilon V(G)\}$, the set of state transitions functions associated with vertex v. The functions assigned to the arcs are

(1) $C = \{C_e : e \epsilon E(G)\}$, the set of *scheduling edge conditions*;

(2) $X = \{X_e : e \epsilon E(G)\}$, the set of *execution edge conditions*;

(3) $T = \{t_e : e \epsilon E(G)\}$, the set of *edge delay times as time intervals*; and

(4) $\Gamma = \{\gamma_e : e \epsilon E(G)\}$, the set of *event execution priorities*.

With the modifications described above, the simulation graph model (SGM) is now defined as $\mathfrak{S} = (F, C, X, T, \Gamma, G)$. This is consistent with Ingalls et al. [1996], except that the edge delay times are now stored as temporal intervals.

As with the framework in Schruben and Yücesan [1993] and Ingalls et al. [1996], this framework also needs additional mechanisms to facilitate the execution of the model. The symbol τ will be used to represent the global simulation clock stored as a time interval and L, the event calendar. However, our definition of L is different from the one specified in Schruben and Yücesan [1993] and Ingalls et al. [1996]. Our definition of L is an ordered set, $L = \{(x_1, \gamma_1, v_1, e_1, a_1, b_1)(x_2, \gamma_l, v_2, e_2, a_2, b_2), \ldots\}$, where x_i represents the execution time, γ_i represents the execution priority, v_i is the vertex to be executed, e_i is the index of the edge that is being scheduled, a_i are the values of the edge attributes, and b_i are the times at which the events are scheduled. Each (x, γ, v, e, a, b) is an event notice. We have added b to the event notice in this paper in order to handle constant delay intervals.

Consistent with Ingalls et al. [1996], we define the following sets:

(1) S_v: the set of state variables that can be modified at vertex v (since S is the set of all state variables, $S_v \subset S$);

(2) P_v: the set of state variables in the parameter list of vertex v;

(3) Φ_e: the set of state variables involved in the scheduling conditions on edge e;

(4) ϑ_e: the set of state variables involved in the execution conditions on edge e;

(5) A_e: the set of state variables used to determine the values of a_e on edge e.

In addition to these sets, we define ω, which represents the termination conditions for the simulation.

To extend this algorithm to include temporal intervals, we must also include two new sets:

(1) H: the set of saved states. This is used to iterate through all possible states in the simulation.

(2) N_h: the NOS, or set of possible next events.

We also define two new variables h and n_h. The variable h is used to count the number of saved states and iterate through the set H; the variable n_h is used to iterate through the N_h set. With these definitions, the execution of the simulation graph model with temporal intervals is carried out as follows.

***Algorithm*.**

Run Initialization
Initialize the saved state set and counter.
New Process Initialization

Step 1. Initialize the global simulation clock. $\tau \leftarrow [0, 0]$.

Step 2. Insert one or more event notices into the event calendar. For simplicity, we assume that each of these event notices could be executed at time [0,0]. $L = L \cup \{([0, 0], \gamma_1, v_1, e_1, a_1, b_1), ([0, 0], \gamma_2, v_2, e_2, a_2, b_2), \ldots\}$.

Execute (execution of the model implementation)

Step 1. Determine the NOS, the set Q is the set of all event notices that could be executed next without regard to priority.

$$Q = \{l \,|\, x_l^- \leq (\min(x_l^+) \forall l \epsilon L) \forall l \epsilon L\}$$

$$N_h = \{q \,|\, \gamma_q = (\min(\gamma_q) \forall q \epsilon Q) \forall q \epsilon Q\}$$

Step 2. If $|N_h| = 1$, then go to Step 6 of execute.

Step 3. Initialize the variable to loop through the NOS. $n_h \leftarrow 1$.

Step 4. Save the state of the simulation by saving the state information in the save-state stack and incrementing the save-state counter. $H_h = \{S, L\}$. $h \leftarrow h + 1$.

Step 5. Remove the $(N_{h-1})_{n_{h-1}}$ event notice from L. $L = L \backslash \{l \backslash l = (N_{h-1})_{n_{h-1}}\}$. Event notice l is removed from the calendar. Go to Step 7 of Execute.

Step 6. Remove the first event notice from L. $L = L \backslash \{l \backslash l = (x_1, \gamma_1, v_1, e_1, a_1, b_1)\}$. Event notice l is removed from the calendar.

Step 7. Evaluate the execution edge condition, $X_{e1}(\vartheta_{e1}, a_1)$. If $X_{e1}(\vartheta_{e1}, a_1) = FALSE$, then go to Step 15 of execute, or else go to Step 8 of Execute.

Step 8. Determine the possible new simulation clock time. $\tau' \leftarrow [\max(x_1^-, \tau^-), \min(x_l^+ \forall l \epsilon L)]$.

Step 9. If t_{e_l} is a constant delay interval, determine if the constant delay time is still valid. It is still valid if $t_{e_l} = \tau' - b_l$. If t_{e_l} is a constant interval and still valid, or if t_{e_l} is an uncertain interval, go to Step 11 of Execute, else go to Step 10 of Execute.

Step 10. Set $t_{e_l} \leftarrow \tau' - b_l$. If $t_{e_l}^- \leq t_{e_l}^+$, then the new t_{e_l} is valid, but the process must be started at the beginning, so that t_{e_l} can be consistent throughout the process. Go to Step 1 of new process initialization. Otherwise, there is no valid constant interval for this process and the process is declared "invalid" and terminates. In that case, go to Step 16 of Execute.

Step 11. Update the simulation clock. $\tau \leftarrow \tau'$.

Step 12. Assign the attributes to the parameters of the vertex. $P_{v_l} \leftarrow a_l$. If Y is the ith state variable in the vertex parameter list, i.e., $(P_v)_i = Y$, then $Y \leftarrow (a_l)_i$.

Step 13. Evaluate the state change. $S_{v_1} \leftarrow f_{v_l}(S)$.

Step 14. Schedule further events. For each edge, e_{lj}, emanating from v_l, if $C_{e_{lj}}(\Phi_{e_l}) = TRUE$, then evaluate $A_{e_{lj}}$ and assign the attribute value of the new event notice, k, $a_k \leftarrow A_{e_{lj}}$. Generate the interevent time, $b_k = f(t_{e_{lj}})$, and schedule the event notice where $L = L \cup \{(\tau + b_k, \gamma_{e_{1j}}, v_j, e_{1j}, a_k, b_k)\}$.

Step 15. If any of the following conditions are satisfied: (i) $\tau > T_{stop}$; (ii) the simulation stopping condition, ω, evaluates TRUE; or (iii) L is empty; then the simulation has reached the end of the process. If $h = 1$, then terminate the simulation.

Step 16. Increment $n_{h-1} \leftarrow n_{h-1} + 1$. If $n_{h-1} \le |N_{h-1}|$ then go to Step 5 of Execute.

Step 17. Restore the last saved system state off the saved-state stack: $h = h - 1$, $L = (L | L \epsilon H_h)$, $S = (S | S \epsilon H_h)$. Go to Step 5 of execute.

4.3 Implementing Execution Processes and the Temporal Interval Calendar

This section describes the core procedures for execution processes and the TIC in QDES as we have implemented them in Smalltalk [Goldberg and Robson 1989]. The software for this implementation is available upon request from the authors. The TIC currently executes a depth-first search on all possible processes. Figure 6 shows the calling tree of the 9 different procedures used in the algorithm. The primary procedure is *Spawn_StartingTime*. This procedure controls the actual execution of events and the spawning of new processes in the simulation. All of the other routines handle relatively simple tasks such as initialization, supervisory control of the program, or the management of the processes that are to be executed later in the program. We provide details for all the procedures in Figure 6 in algorithmic format.

There are several variables used in the algorithm. Their definitions are as follows:

(1) *calendar*: the current event for the simulation;

(2) *calendar.nextTime*: the scheduled time of the first event notice on the event calendar;

(3) *calendar.time*: the current time of the simulation;

(4) *calendarStack*: the stack of event calendars for the processes that are being created;

(5) *event*: the current event notice that is to be executed;

(6) *event.attributes*: the attributes of the event notice;

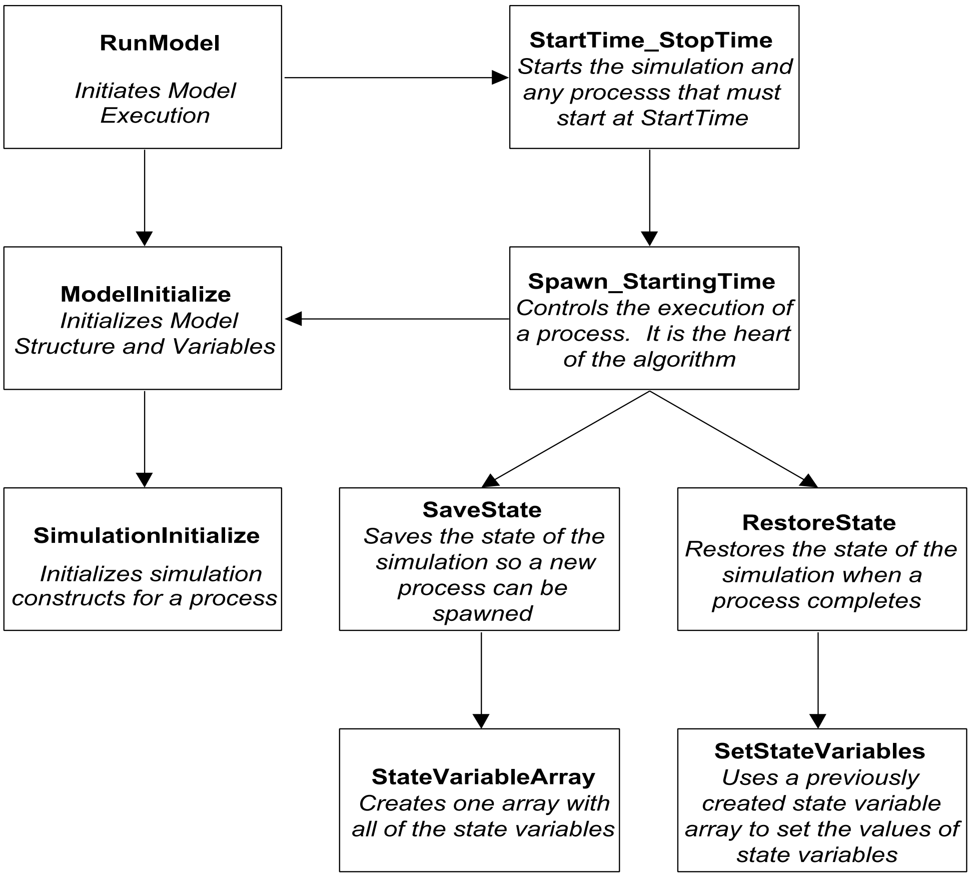
RunModel
Initiates Model Execution
StartTime_StopTime
Starts the simulation and any processs that must start at StartTime
ModelInitialize
Initializes Model Structure and Variables
Spawn_StartingTime
Controls the execution of a process. It is the heart of the algorithm
SimulationInitialize
Initializes simulation constructs for a process
SaveState
Saves the state of the simulation so a new process can be spawned
RestoreState
Restores the state of the simulation when a process completes
StateVariableArray
Creates one array with all of the state variables
SetStateVariables
Uses a previously created state variable array to set the values of state variables

Fig. 6. Calling tree of the algorithm.

(7) *event.executionCondition*: a conditional that if evaluated TRUE, then the event notice executes;

(8) *event.node*: the vertex to be executed by the event notice;

(9) *executeEvent*: a boolean that gives the result of an execution condition;

(10) *firstEvent*: the first event notice scheduled on each new process;

(11) *firstEventFlag*: if TRUE, then the firstEvent is the event notice to be executed. If FALSE, then get the nondeterministically ordered set from the event calendar;

(12) *model.constantDelayInterval()*: an array of constant delay intervals. It is the current value of that constant delay interval in this process;

(13) *modelProcess*: the current model process;

(14) *newStartTime*: the time a spawned process will start;

(15) *nextTime*: the time of the simulation if a given event notice is executed;

(16) *possibleEvents*: the set of event notices that can be executed next (the nondeterministically ordered set);

(17) *possibleEvents().constantDelayInterval*: if a delay for an event notice is a constant interval, this is a set to the current value of that constant interval;

(18) *possibleEvents().scheduledTime*: the time this event notice was scheduled (i.e., put on the event calendar);

(19) *possibleEvents().time*: the time the event notice is scheduled to occur;

(20) *startTimeValue*: the user-defined beginning of the simulation.

(21) *stateStack*: the stack of state variable values for processes that are being created;

(22) *stoppingCondition*: a conditional that if evaluated to TRUE, then the simulation stops;

(23) *stopTimeValue*: the user defined end of the simulation.

The procedure *RunModel* is executed by the user. Its primary function is to call procedures that initialize the model and then pass control to the procedures that handle the execution of the model.

```
Procedure RunModel
    Call ModelInitialize
    Call StartTime_StopTime(startTimeValue,stopTimeValue)
End Procedure RunModel
```

The procedure *ModelInitialize* handles the initialization of any process that must start at the beginning of the simulation. There are two types of processes that fall into this category. One is the first process of the simulation. The other type is a process that is spawned because a constant interval is found to be invalid. In the latter case, a new value for the constant interval is determined and a new process is created from the beginning of the simulation to see if the new constant interval value can remain valid throughout the simulation. *ModelInitialize* also handles the initialization needed for the specific model being executed.

```
Procedure ModelInitialize
    Call SimulationInitialize
    Initialize state variables
    Set stoppingCondition
    Designate the first vertex (or vertices) to be executed
End Procedure ModelInitialize
```

The procedure *SimulationInitialize* is a generic procedure that handles the initialization of the simulation structures of any qualitative model. If a random number generator is required in the simulation, it is initialized.

This procedure also initializes the constructs needed to handle multiple processes.

```
Procedure SimulationInitialize
    Initialize the random number generator
    If (calendarStack has not been initialized) Then
        Initialize calendarStack, stateStack, modelProcess
    End If
End Procedure SimulationInitialize
```

The procedure *StartTime_StopTime* seeds the calendar with the proper information and calls *Spawn_StartingTime*, which controls the process execution.

```
Procedure StartTime_StopTime(startTimeValue,stopTimeValue)
    calendar.time:= startTimeValue
    Seed the calendar with the first node (or nodes) to be executed
    firstEventFlag:= TRUE.
    Call Spawn_StartingTime(the first event on the calendar, startTimeValue)
End Procedure StartTime_StopTime
```

Because of its primacy and complexity, a flowchart for the procedure *Spawn_StartingTime* is given in Figure 7. *Spawn_StartingTime* is the procedure that takes a process to completion. Once a new process is spawned, *Spawn_StartingTime* will handle all of the control of that process. *Spawn_StartingTime* will terminate under two conditions. The first condition occurs when a process finishes under normal simulation stopping conditions such as the calendar being empty or the simulation being past the ending simulation time. The second termination condition occurs when the process has a constant interval that cannot be adjusted so that it becomes valid. Both of these termination conditions for the procedure *Spawn_StartingTime* would not be considered exceptions in any way.

Spawn_StartingTime has two arguments that are passed to it. The first, *firstEvent*, a pointer to the first event that will be executed in the spawned process. The second, *startTimeValue*, is the clock time for that event execution. Other comments for this procedure will be in normal typeface.

```
Procedure Spawn_StartingTime(firstEvent, startTimeValue)
The procedure is a large loop that checks end-of-simulation conditions at the bottom.
    Do
        If (firstEventFlag = TRUE) Then
        If the firstEventFlag is true, then it has been predetermined that firstEvent
        will be the next event executed.
        event:= firstEvent
            Remove firstEvent from the event calendar
            firstEventFlag:= FALSE
        Else
        If firstEventFlag is false, then we must determine which event notices could
        possibly be executed next.
            possibleEvents:= set of event notices that can execute next
            If (|possibleEvents| = 1) Then
                If (possibleEvents(1).time ⊆ calender.time) Then
                    nextTime:= possibleEvents(1).time
                Else
                    nextTime:= max(possibleEvents(1).time,calendar.time)
```

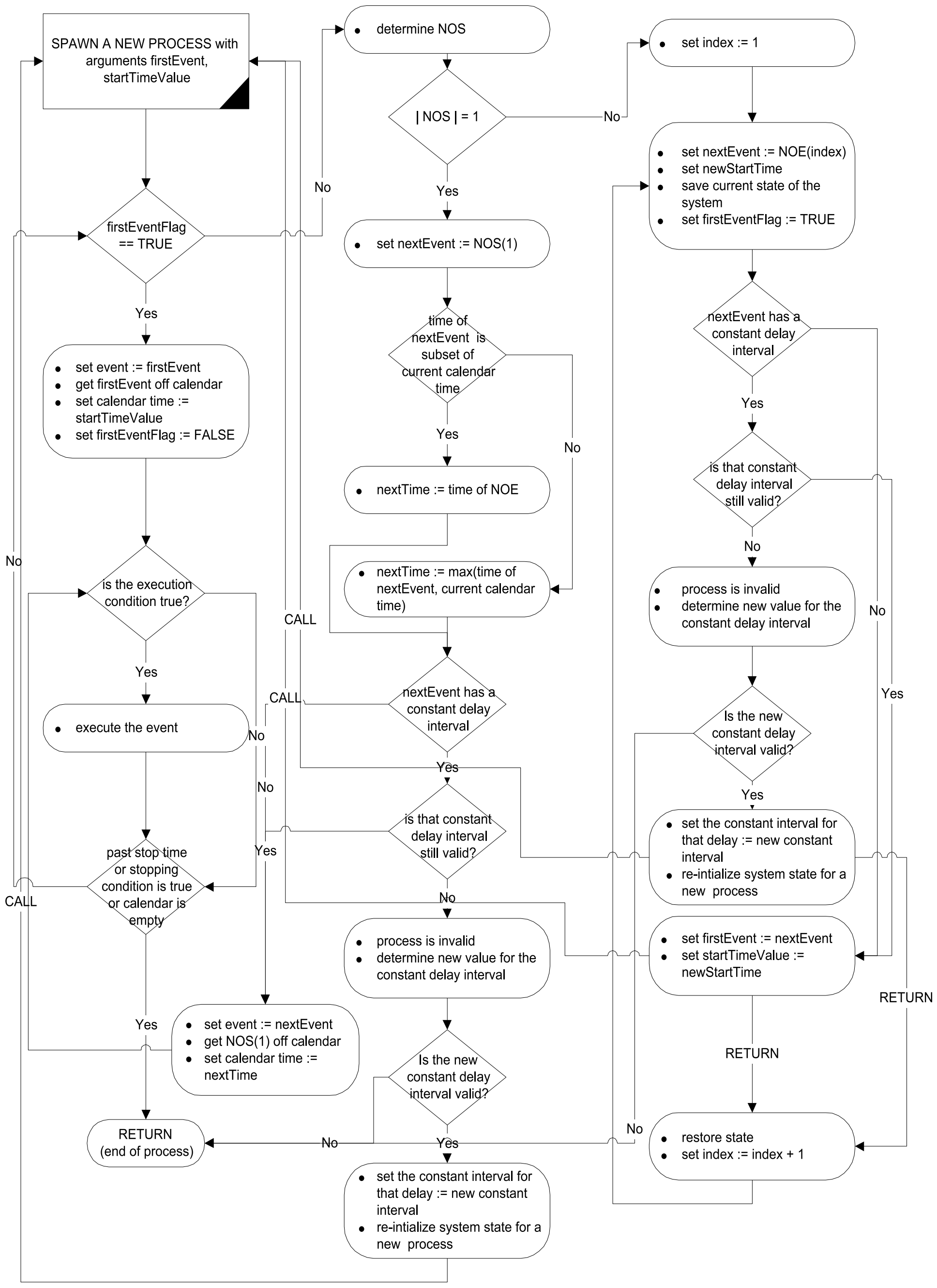
SPAWN A NEW PROCESS with arguments firstEvent, startTimeValue
firstEventFlag == TRUE
Yes
No
set event := firstEvent
get firstEvent off calendar
set calendar time := startTimeValue
set firstEventFlag := FALSE
is the execution condition true?
Yes
execute the event
No
past stop time or stopping condition is true or calendar is empty
Yes
CALL
RETURN (end of process)
determine NOS
| NOS | = 1
Yes
set nextEvent := NOS(1)
time of nextEvent is subset of current calendar time
Yes
No
nextTime := time of NOE
nextTime := max(time of nextEvent, current calendar time)
nextEvent has a constant delay interval
CALL
is that constant delay interval still valid?
No
process is invalid
determine new value for the constant delay interval
Is the new constant delay interval valid?
set the constant interval for that delay := new constant interval
re-intialize system state for a new process
set event := nextEvent
get NOS(1) off calendar
set calendar time := nextTime
set index := 1
set nextEvent := NOE(index)
set newStartTime
save current state of the system
set firstEventFlag := TRUE
nextEvent has a constant delay interval
is that constant delay interval still valid?
Is the new constant delay interval valid?
set firstEvent := nextEvent
set startTimeValue := newStartTime
RETURN
restore state
set index := index + 1

Fig. 7. A flow chart of the logic for the procedure Spawn_StartingTime.

```
End If
If (possibleEvents(1).constantDelayInterval exists) Then
    Check to see if constant interval is still valid.
    If (possibleEvents(1).constantDelayInterval = nextTime –
    possibleEvents(1).scheduledTime) Then
```

```
        The constant interval is valid.
            event:= possibleEvents(1)
            Remove possibleEvents(1) from the calendar
        Else
        The constant interval is not valid. A new process must be created
        with a valid value for the constant interval, if one can be found.
            Call ModelInitialize
            x:= nextTime – possibleEvents(1).scheduledTime
            If (x is valid) Then
                i:= corresponding model.constantDelayInterval
                model.constantDelayInterval(i):= x
                modelProcess:= modelProcess + 1
                Call StartTime_StopTime(startTimeValue, stopTimeValue)
            End If
            Return
        End If
    End If
Else
If there is more than one event notice in the possibleEvents set, then the
model creates a process for each event notice in the set and assumes that
the event notice is the next event notice to be executed.
    For (index = 1 to |possibleEvents|)
        Set newStartTime
        Call SaveState
        firstEventFlag:= TRUE
        If (possibleEvents(index).constantDelayInterval exists) Then
            If (possibleEvents(index).constantDelayInterval isValid) Then
                Call Spawn_StartingTime(possibleEvents(index), newStart-
                Time)
                Call RestoreState
            Else
                Call ModelInitialize
                x:= newStartTime – possibleEvents(index).scheduledTime
                If (x is valid) Then
                    i:= corresponding model.constantDelayInterval
                    model.constantDelayInterval(i):= x
                    modelProcess:= modelProcess + 1
                    Call StartTime_StopTime(startTimeValue, stopTimeValue)
                End If
                Call RestoreState
            End If
        Else
            Call Spawn_StartingTime(possibleEvents(index), newStartTime)
            Call RestoreState
        End If
    End For
    Return
    End If
End If
If you have gotten to this point, then there is actually an event notice to
execute. The first thing to do is to see if its execution is true.
executeEvent:= event.executionCondition
If (executeEvent = TRUE) Then
If the execution condition is true, then execute the code that is associated
with the vertex that is to be executed. The event.vertex procedure is
model-dependent, but would contain logic for changing state variables
```

```
      and/or scheduling event notices.
        Call event.node(event.attributes)
      End If
      eventNumber:= eventNumber + 1
    Here, at the bottom of the procedure loop, we check to see if the process has
    come to an end. It has not come to an end if the event calendar is not empty
    and the execution time of the first event notice on the event calendar is less
    than or equal to the ending time of the simulation and the evaluation of the
    stopping condition is false.
    While (calendar is not empty & calendar.nextTime <= stopTime & stopping-
    Condition has not been reached)
  End Procedure Spawn_StartingTime
```

The procedure *SaveState* is used to save the current state of the simulation. The current event calendar is added to the *calendarStack*, and the current values of the state variables are added to the *stateStack*. Note that the "+" sign in this procedure denotes adding an element to the end of an ordered collection.

```
Procedure SaveState
  calendarStack:= calendarStack + calendar
  stateStack:= stateStack + Call StateVariableArray
End Procedure StaveState
```

The procedure *RestoreState* is used to restore a previously saved state of the simulation. The *calendar* is set to the last event calendar in the *calendarStack*, and the state variables are set to the values held in the last array in the *stateStack*.

```
Procedure RestoreState
  If (calendarStack ≠ ∅) Then
    calendar:= last calendar in calendarStack
    calendarStack:= calendarStack – last calendar in calendarStack
  End If
  If (stateStack ≠ ∅) Then
    Call SetStateVariables(last stateVariableArray in stateStack)
    stateStack:= stateStack – last stateVariableArray in stateStack
  End If
End Procedure RestoreState
```

The procedure *StateVariableArray* creates an array with the values of all of the state variables in the model.

```
Procedure StateVariableArray
  Return an array with the current value of all of the state variables
End Procedure StateVariableArray
```

The procedure *SetStateVariables* takes a previously saved *stateVariable-Array* and sets the state variables in the simulation to the values held in that array.

```
Procedure SetStateVariables(stateVariableArray)
  Assign the values of stateVariableArray to the corresponding state variables
End Procedure SetStateVariables
```

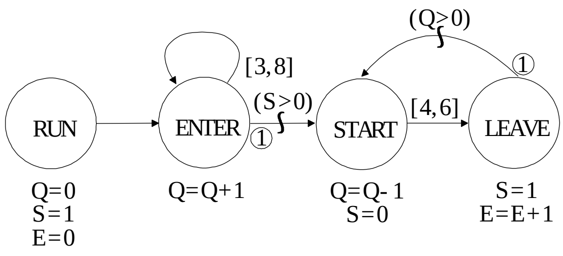

Fig. 8. The qualitative example.

5. AN EXAMPLE

As an example, consider the EG of a firm that receives supplier shipments, processes them one shipment at a time, and then sends the shipments to customers. This is an example of the standard inbound-processing-outbound description of a firm used in logistics and supply chain management (e.g., Coyle et al. [1992, chap. 3]. We assume that there are no inventory capacity constraints. Consequently, this system can be modeled using the basic single server queueing model found in Schruben [1995, chap. 2]. We consider this model for two reasons: First, it illustrates that QDES has the same functionality of RDES. Second, it underscores our recommendation that QDES be used to model problems at a more strategic level than at lower operations level. The model applies at a relatively high level (i.e., the level of the firm, as opposed to level of the shop floor) where event granularity is high, and precise intershipment and processing data may not be available and/or very difficult to obtain. Figure 8 contains an EG model of this system. The node labeled RUN is the first node executed in the system. It is used to initialize the state variables. The node labeled ENTER represents the event that a shipment arrives to the system. The nodes START and LEAVE represent the following events: begin processing the shipment and end processing the shipment, respectively. The state variables are Q for the number of shipments waiting for processing, S for the status of the processing capacity, and E for the number of shipments that have exited. If S=1, then the processing capacity is idle; if S=0, then the processing capacity is busy. Since changes in the state variables happen only when an event occurs, the changes are stated below each event. For example, when the START processing event occurs, the queue of waiting shipments decrements by one (Q=Q-1) and the processing capacity status changes to busy (S=0).

The conditional expressions that appear on some of the edges are based on the state of the system. For example, the condition between the ENTER event and the START event is a condition to test if the processing capacity is idle (S>0). In other words, if a shipment arrives and the processing capacity is not busy, then the shipment enters processing immediately. It should also be noted that EG edges can have priorities. This model has high priorities on the edge between ENTER and START and the edge between LEAVE and START. All of the other edges use the lower default priority of

Process	Calendar Time	Event Executed	S	Q	E	Future Events (Time,Node,Pri)	Process	Calendar Time	Event Executed	S	Q	E	Future Events (Time,Node,Pri)
1	[0,0]	RUN	1	0	0	[0,0],ENTER,9							
1	[0,0]	ENTER	1	1	0	[0,0],START,1 [3,8],ENTER,9							
1	[0,0]	START	0	0	0	[3,8],ENTER,9 [4,6],LEAVE,9							
1	[3,6]	ENTER	0	1	0	[4,6],LEAVE,9 [6,14],ENTER,9	520	[4,6]	LEAVE	1	0	1	[3,8],ENTER,9
1	[4,6]	LEAVE	1	1	1	[4,6],START,1 [6,14],ENTER,9	520	[4,8]	ENTER	1	1	1	[4,8],START,1 [7,16],ENTER,9
1	[4,6]	START	0	0	1	[6,14],ENTER,9 [8,12],LEAVE,9	520	[4,8]	START	0	0	1	[7,16],ENTER,9 [8,14],LEAVE,9
1	[6,12]	ENTER	0	1	1	[8,12],LEAVE,9 [9,20],ENTER,9	520	[7,14]	ENTER	0	1	1	[8,14],LEAVE,9 [10,22],ENTER,9
1	[8,12]	LEAVE	1	1	2	[8,12],START,1 [9,20],ENTER,9	520	[8,14]	LEAVE	1	1	2	[8,14],START,1 [10,22],ENTER,9
1	[8,12]	START	0	0	2	[9,20],ENTER,9 [12,18],LEAVE,9	520	[8,14]	START	0	0	2	[10,22],ENTER,9 [12,20],LEAVE,9
1	[9,18]	ENTER	0	1	2	[12,18],LEAVE,9 [12,26],ENTER,9	520	[10,20]	ENTER	0	1	2	[12,20],LEAVE,9 [13,28],ENTER,9
1	[12,18]	LEAVE	1	1	3	[12,18],START,1 [12,26],ENTER,9	520	[12,20]	LEAVE	1	1	3	[12,20],START,1 [13,28],ENTER,9
1	[12,18]	START	0	0	3	[12,26],ENTER,9 [16,24],LEAVE,9	520	[12,20]	START	0	0	3	[13,28],ENTER,9 [16,26],LEAVE,9
1	[12,24]	ENTER	0	1	3	[15,32],ENTER,9 [16,24],LEAVE,9	520	[13,26]	ENTER	0	1	3	[16,26],LEAVE,9 [16,34],ENTER,9
1	[15,24]	ENTER	0	2	3	[16,24],LEAVE,9 [18,32],ENTER,9	520	[16,26]	LEAVE	1	1	4	[16,26],START,1 [16,34],ENTER,9
1	[16,24]	LEAVE	1	2	4	[16,24],START,1 [18,32],ENTER,9	520	[16,26]	START	0	0	4	[16,34],ENTER,9 [20,32],LEAVE,9
1	[16,24]	START	0	1	4	[18,32],ENTER,9 [20,30],LEAVE,9	520	[16,32]	ENTER	0	1	4	[19,40],ENTER,9 [20,32],LEAVE,9
1	[18,30]	ENTER	0	2	4	[20,30],LEAVE,9 [21,38],ENTER,9	520	[19,32]	ENTER	0	2	4	[20,32],LEAVE,9 [22,40],ENTER,9
1	[20,30]	LEAVE	1	2	5	[20,30],START,1 [21,38],ENTER,9	520	[20,32]	LEAVE	1	2	5	[20,32],START,1 [22,40],ENTER,9

Fig. 9. TIC processes example with uncertain intervals.

9. Using the example model in Figure 8, we want to illustrate how TIC works by simulating five shipment-processing completions (LEAVE events).

5.1 Simulation Execution

5.1.1 *Example with Uncertain Intervals*. In the first run of this example, we assume that the two delay times (in days), *[3,8]* and *[4,6]* are uncertain intervals. This example tracks the first process in the system to completion (see Figure 9). Process 520 is created when the model cannot determine the ordering for the two event notices on the event calendar: *[3,8],* ENTER,9 and *[4,6],* LEAVE,9. These two event notices have event times that intersect (*[3,8]* and *[4,6]*) and have the same priority (9). Process 1 continues as if *[3,8],* ENTER,9 was the first event notice, while process 520 is placed on a stack and will be executed at a later time. When process 520 is removed from the stack, it is the 520th process to be executed, and it continues as if *[4,6],* LEAVE,9 was the first event notice. The remaining nondeterministic orderings found in processes 1 and 520 help create the 897 processes created during the simulation run.

Figure 9 shows the two processes, 1 and 520, described above. Figure 9 gives examples of several qualitative calendar mechanisms at work. For Process 1 at time *[0,0]*, we find two event notices on the calendar with intersecting scheduled execution times. As described above, process 1 chooses the *[3,8]*, ENTER,9 event notice to execute first. However, the simulation clock is set to *[3,6]*. This is because if the event notice *[3,8],* ENTER,9 executes before the event notice *[4,6]*, LEAVE,9, the simulation clock cannot be earlier than time 3 and cannot be later than time 6.

Another construct that is shown in Figure 9 is the use of priorities on certain event notices. In this example, we put high priority on the edges with *[0,0]* delay times. The high priority ensures that the scheduled event notice will be executed immediately. This is an extension of the convention of assigning high priorities to zero delay edges recommended by Schruben [1995, p. 121] for EGs to QDES. For example, in process 1 at time *[8,12]* when the LEAVE event notice has just been executed, we see two event notices, *[8,12]*, START,1 and *[9,20]*, ENTER,9. Even though the time intervals overlap on these two event notices, the high priority event notice (*[8,12],* START,1) will always be executed first. Using priorities in this manner does not violate the logic of NOE, since if the LEAVE event notice occurs before the ENTER event notice, then the START event notice must occur before the ENTER event notice as well, since LEAVE and START occur at the same time. The only point of concern is if all three event notices are scheduled to occur at the same time, in which case priorities are being used in exactly the same way as suggested by Schruben [1995].

5.1.2 *Example with a Constant Interval.* Using the same model, we change the arrival interval to a constant interval with the intial value *[3,8]*. The change means that the arrival interval must be constant throughout the entire process. When the interval *[3,8]* is no longer valid for a process, the simulation adjusts the constant interval so that it will be valid, and starts a new process from time *[0,0]*. When the events *[3,8]*, ENTER,9 and *[4,6],* LEAVE,9 are on the calendar and process 1 chooses to execute *[3,8],* ENTER,9 first, the constant interval *[3,8]* can no longer be valid. The reason is that the simulation clock when *[3,8],* ENTER,9 is executed will be at *[3,6]*, not *[3,8]*. At this point the process is declared invalid, the constant interval for the arrival time is changed from *[3,8]* to *[3,6]*, and a new process is started at time *[0,0]* with the initial simulation conditions. This type of adjustment of the constant interval continues until it does not have to change for an entire process. When that is the case, the process is valid for that particular value of the constant interval.

Figure 10 shows the trace of the simulation as it obtains its first valid process. The headings in Figure 10 are as follows:

(1) *Process* is the number of the process in the simulation.

(2) *Event* is the number of the event in the simulation.

(3) *Spawning Event* is the event that spawned this process or part of a process.

(4) *Calendar Time* is the current simulation time.

(5) *Event Executed* is the name of the event executed.

```
Process Event Spawning Event Calendar Time Event Executed S Q E Arrival Interval Calendar Event 1      Calendar Event 2
   1      1       nil          [0,0]          RUN        1 0 0 C[3,8]           [0,0],ENTER,5
   1      2       nil          [0,0]          ENTER      1 1 0 C[3,8]           [0,0],START,1          [3,8],ENTER,5
   1      3       nil          [0,0]          START      0 0 0 C[3,8]           [3,8],ENTER,5          [4,6],LEAVE,5
*** CONSTANT DELAY C[3,8] INVALID: EVENT [3,8],ENTER,5 WAS SCHEDULED AT TIME [0,0], CLOCK TIME IS [3,6] ***
   1      4       nil          [0,0]          RUN        1 0 0 C[3.0,6.0]       [0,0],ENTER,5
   1      5       nil          [0,0]          ENTER      1 1 0 C[3.0,6.0]       [0,0],START,1          [3.0,6.0],ENTER,5
   1      6       nil          [0,0]          START      0 0 0 C[3.0,6.0]       [3.0,6.0],ENTER,5      [4,6],LEAVE,5
   1      7       nil          [3.0,6.0]      ENTER      0 1 0 C[3.0,6.0]       [4,6],LEAVE,5          [6.0,12.0],ENTER,5
   1      8       nil          [4.0,6]        LEAVE      1 1 1 C[3.0,6.0]       [4.0,6],START,1        [6.0,12.0],ENTER,5
   1      9       nil          [4.0,6]        START      0 0 1 C[3.0,6.0]       [6.0,12.0],ENTER,5     [8.0,12],LEAVE,5
   1     10       nil          [6.0,12.0]     ENTER      0 1 1 C[3.0,6.0]       [8.0,12],LEAVE,5       [9.0,18.0],ENTER,5
   1     11       nil          [8.0,12]       LEAVE      1 1 2 C[3.0,6.0]       [8.0,12],START,1       [9.0,18.0],ENTER,5
   1     12       nil          [8.0,12]       START      0 0 2 C[3.0,6.0]       [9.0,18.0],ENTER,5     [12.0,18],LEAVE,5
   1     13       nil          [9.0,18.0]     ENTER      0 1 2 C[3.0,6.0]       [12.0,18],LEAVE,5      [12.0,24.0],ENTER,5
   1     14       nil          [12.0,18]      LEAVE      1 1 3 C[3.0,6.0]       [12.0,18],START,1      [12.0,24.0],ENTER,5
   1     15       nil          [12.0,18]      START      0 0 3 C[3.0,6.0]       [12.0,24.0],ENTER,5    [16.0,24],LEAVE,5
   1     16       nil          [12.0,24.0]    ENTER      0 1 3 C[3.0,6.0]       [15.0,30.0],ENTER,5    [16.0,24],LEAVE,5
*** CONSTANT DELAY C[3,6] INVALID: EVENT [15,30],ENTER,5 WAS SCHEDULED AT TIME [12,24], CLOCK TIME IS [15,24] ***
   1     17       16           [16.0,24]      LEAVE      1 1 4 C[3.0,6.0]       [15.0,30.0],ENTER,5    [16.0,24],START,1
   1     18       16           [16.0,24]      START      0 0 4 C[3.0,6.0]       [15.0,30.0],ENTER,5    [20.0,30],LEAVE,5
*** CONSTANT DELAY C[3,6] INVALID: EVENT [15,30],ENTER,5 WAS SCHEDULED AT TIME [12,24], CLOCK TIME IS [16,30] ***
   1     19       16           [0,0]          RUN        1 0 0 C[4.0,6.0]       [0,0],ENTER,5
   1     20       16           [0,0]          ENTER      1 1 0 C[4.0,6.0]       [0,0],START,1          [4.0,6.0],ENTER,5
   1     21       16           [0,0]          START      0 0 0 C[4.0,6.0]       [4.0,6.0],ENTER,5      [4,6],LEAVE,5
   1     22       16           [4.0,6.0]      ENTER      0 1 0 C[4.0,6.0]       [4,6],LEAVE,5          [8.0,12.0],ENTER,5
   1     23       16           [4,6]          LEAVE      1 1 1 C[4.0,6.0]       [4,6],START,1          [8.0,12.0],ENTER,5
   1     24       16           [4,6]          START      0 0 1 C[4.0,6.0]       [8.0,12.0],ENTER,5     [8,12],LEAVE,5
   1     25       16           [8.0,12.0]     ENTER      0 1 1 C[4.0,6.0]       [8,12],LEAVE,5         [12.0,18.0],ENTER,5
   1     26       16           [8.0,12]       LEAVE      1 1 2 C[4.0,6.0]       [8.0,12],START,1       [12.0,18.0],ENTER,5
   1     27       16           [8.0,12]       START      0 0 2 C[4.0,6.0]       [12.0,18.0],ENTER,5    [12.0,18],LEAVE,5
   1     28       16           [12.0,18.0]    ENTER      0 1 2 C[4.0,6.0]       [12.0,18],LEAVE,5      [16.0,24.0],ENTER,5
   1     29       16           [12.0,18]      LEAVE      1 1 3 C[4.0,6.0]       [12.0,18],START,1      [16.0,24.0],ENTER,5
   1     30       16           [12.0,18]      START      0 0 3 C[4.0,6.0]       [16.0,24.0],ENTER,5    [16.0,24],LEAVE,5
   1     31       16           [16.0,24.0]    ENTER      0 1 3 C[4.0,6.0]       [16.0,24],LEAVE,5      [20.0,30.0],ENTER,5
   1     32       16           [16.0,24]      LEAVE      1 1 4 C[4.0,6.0]       [16.0,24],START,1      [20.0,30.0],ENTER,5
   1     33       16           [16.0,24]      START      0 0 4 C[4.0,6.0]       [20.0,30.0],ENTER,5    [20.0,30],LEAVE,5
   1     34       16           [20.0,30.0]    ENTER      0 1 4 C[4.0,6.0]       [20.0,30],LEAVE,5      [24.0,36.0],ENTER,5
   1     35       16           [20.0,30]      LEAVE      1 1 5 C[4.0,6.0]       [20.0,30],START,1      [24.0,36.0],ENTER,5
*** VALID PROCESS ***
```

Fig. 10. Constant interval arrival Time Trace.

(6) *S* is the state variable value of the server. *S* is the number of available servers.

(7) *Q* is the state variable value for the number in the queue.

(8) *E* is the state variable value for the number of exits.

(9) *Arrival Interval* is the current value of the constant interval for the interarrival time.

(10) *Calendar Event 1* and *Calendar Event 2* are the event notices on the event calendar. Each event notice is shown in a triple with the scheduled execution time for the event notice, the event to be executed, and the event notice priority.

The initial condition for the constant interval, called arrival interval, is *[3,8]*. The first time that the arrival interval is declared invalid is at time *[0,0]* because the new schedule time would be *[3,6]* and the execution time of the ENTER event is *[3,8]*. At this point, the arrival interval is adjusted to *[3,6]* and the process is restarted at time *[0,0]*. At Event 16, the interval

[3,6] is declared invalid because the new clock time is *[15,24]* and the ENTER event was scheduled to execute at *[15,30]*. This event notice was put on the event calendar at time *[12,24]*. In order to have this process be restarted with a constant interval, the new constant interval would need to be *[15,24] – [12,24] = [3,0]*, which is invalid. At this point, the simulation knows that there cannot be a restart at time *[0,0]*, and opts instead to take an event notice off the stack and try to make it work. When Event 16 was executed, there were two events notices on the event calendar, *[15,30]*, ENTER,9 and *[16,24]*, LEAVE,5. The previous attempts assumed that *[15,30]*, ENTER,9 was executed first. At this point, the simulation assumes that *[16,24]*, LEAVE,5 is executed first. At Event 18, there is an adjustment of the constant arrival interval from *[3,6]* to *[4,6]* and the process is restarted at time *[0,0]*. Events 19-35 constitute the first valid process in the simulation. In this sequence of events, the constant arrival time lies in the interval *[4,6]* and this process can be valid for the simulation.

5.2 Temporal Interval Outputs

When temporal intervals are used in DES, the results can be analyzed by examining the timing of events or conditions. For example, Figure 9 illustrates that the fifth shipment exits in interval *[20,30]* for process 1 and *[20,32]* for process 520. Using all possible processes, one can determine that the fifth exit of the system must occur in the interval *[20,38]*. Interpreting the inputs to the model, we can say that regardless of the distribution placed on the arrival time interval of *[3,8]* and the service time interval of *[4,6]*, the fifth exit of the system will occur in the interval *[20,38]*. For example, we ran 1000 replications of a quantitative model where the input distributions were uniform over *[3,8]* and *[4,6]*, and the fifth exit occurred between 21.36 and 35.76 in those experiments.

In the constant interval example, we can see that the first full process that completed as a valid process had a constant arrival interval of *[4,6]*. Overall, there were 98 valid processes generated. Of these valid processes, the constant arrival intervals *[4,6]*, *[5,6]*, and *[6,6]* accounted for 32 processes each, and the constant arrival intervals *[3,6]* and *[4,8]* accounted for 1 process each. For each of the constant intervals in the output, the interval for the fifth exit from the system is given in Figure 11.

The constant interval analysis may be valuable in this example for determining the feasibility of scheduling production activities in the face of various shipping options. For example, suppose that a contract could be signed to guarantee a regular fixed shipping interarrival time. Additionally, suppose that the choices for the shipping interarrival time ranged from 3 days to 8 days. The constant interval simulation analysis could be used to generate feasible processes like those given in Figure 11. In turn, this information could be used as the basis for developing a schedule for processing shipments based on the fixed shipping interval option selected. It is important to note that each constant interval option covers multiple shipping options simultaneously. In addition, the interval analysis also

Constant Interval	Fifth Exit Interval
[3,6]	[20,30]
[4,6]	[20,30]
[5,6]	[24,30]
[6,6]	[28,30]
[4,8]	[20,38]

Fig. 11. Fifth interval exit times for shipments when constant intervals are used for shipment interarrival times.

provides information on the amount of permissible deviation in the schedule. For example, if the shipping interarrival time is set to every six days, Figure 11 indicates that the process based on constant interval *[4,8]* remains feasible if the interarrival time happens to decrease or increase by two days.

6. CONCLUSION

This paper incorporates temporal intervals into DES to produce a QDES approach. This is a significant extension of DES because temporal intervals can be used to model uncertainty in the timing of events, rather than assuming the timing of events is random and using probability distributions. As a result, QDES is a robust form of DES because it requires fewer, possibly erroneous, modeling assumptions and it covers all possible outcomes.

The coverage property ensures that all possible processes are characterized. For planning and scheduling problems, schedules would not have to be rerun every time something did not happen according to plan. Coverage would guarantee that as long as the input intervals are respected, at least one process in the run would characterize the events that had taken place and would give information on the next scheduling decision.

The coverage property could be leveraged for debugging simulation models. Again, a benefit of the coverage property is that anything that would happen in the "normal" discrete event model is characterized in the qualitative model. One could check state variable values to see if they are in valid ranges in every process. Additionally, low probability execution sequences would be executed in the qualitative model. This would give the modeler absolute confidence in the validity of the simulation model.

QDES provides a framework for simulation optimization that has not previously existed. Given a given state variable value on qualitative processes, we can track which processes have the best values and possibly structure optimization models around the sequences represented in those processes. Ingalls [1999] extends QDES by implementing qualitative interval state variables and qualitative conditionals. This extension yields a complete modeling approach for qualitative SGMs. When complete, this methodology will provide a viable approach for modeling of discrete event systems with uncertain input data. Ingalls [1999] develops certain process scoring methodologies to assist in the analysis of data generated by the QDES model. However, more work needs to be done in this area. With the ability to rank or score processes, algorithms can be put in place to separate the more likely processes from the less likely processes. Ingalls [1999] also considers the application of QDES to project management. Project management is a fruitful application area because it is strategic in nature, and it is often difficult to get data for precise estimates of distributions for activity times.

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of the CBA/GSB Faculty Research Committee, College of Business Administration, University of Texas at Austin. Additionally, we are grateful for the improvements suggested by area editor Osman Balci and two anonymous referees.

REFERENCES

ALLEN, J. F. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM 26*, 11 (Nov.), 832–843.

BALANKRISHNAN, A. AND WHINSTON, A. B. 1991. Information issues in model specification. *Inf. Syst. Res. 2*, 4 (Dec.), 263–286.

BONDY, J. A. AND MURTY, U. S. R. 1976. *Graph Theory with Applications*. Macmillan Press Ltd., London, UK.

COYLE, J. J., BARDI, E. J., AND LANGLEY, C. J. JR. 1992. *The Management of Business Logistics*. West Publishing Co., St. Paul, MN.

FORBUS, K. D. 1990. Qualitative physics: past present and future. In *Readings in Qualitative Reasoning about Physical Systems*, D. S. Weld and J. d. Kleer, Eds. Morgan Kaufmann Publishers Inc., San Francisco, CA, 11–39.

GOLDBERG, A. AND ROBSON, D. 1989. *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA.

HINKKANEN, A., LANG, K. R., AND WHINSTON, A. B. 1993. A theoretical foundation of qualitative reasoning based on set theory. Tech. Rep.. Center for Numerical Analysis, The Univ. of Texas at Austin, Austin, TX.

INGALLS, R. G. 1999. Qualitative simulation graph methodology and implementation. Ph.D. Dissertation. Center for Numerical Analysis, The Univ. of Texas at Austin, Austin, TX.

INGALLS, R. G., MORRICE, D. J., AND WHINSTON, A. B. 1996. Eliminating canceling edges from the simulation graph model methodology. In *Proceedings of the 1996 Winter Conference on Simulation* (WSC '96, Coronado, CA, Dec. 8–11), J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, Eds. ACM Press, New York, NY, 825–832.

INGALLS, R. G., MORRICE, D. J., AND WHINSTON, A. B. 1994. Interval time clock implementation for qualitative event graphs. In *Proceedings of the 1994 Winter Conference on Simulation* (WSC '94, Lake Buena Vista, FL, Dec. 11–14), M. S. Manivannan and J. D. Tew, Eds. Society for Computer Simulation, San Diego, CA, 574–580.

KIANG, M. Y., HINKKANEN, A., AND WHINSTON, A. B. 1995. An interval propagation method for solving qualitative difference equations. *IEEE Trans. Syst. Man Cybern. 25*, 1110–1120.

KOUVELIS, P. AND YU, G. 1997. *Robust Discrete Optimization and Its Applications*. Kluwer Academic Publishers, Hingham, MA.

KUIPERS, B. 1987. Qualitative simulation as causal explanation. *IEEE Trans. Syst. Man Cybern. 17*, 3, 432–444.

LAWLER, E. L. 1976. *Combinatorial Optimization: Networks and Matroids*. Holt Rinehart & Winston, Inc./School Division, Austin, TX.

LUCE, R. D. AND RAIFFA, H. 1954. *Games and Decisions: Introduction and Critical Survey*. John Wiley and Sons, Inc., New York, NY.

NANCE, R. E. 1981. The time and state relationships in simulation modeling. *Commun. ACM 24*, 4 (Apr.), 173–179.

SCHRUBEN, L. W. 1995. *Graphical Simulation Modeling and Analysis: Using Sigma for Windows*. Boyd and Fraser Publishing Co., Danvers, MA.

SCHRUBEN, L. 1983. Simulation modeling with event graphs. *Commun. ACM 26*, 11 (Nov.), 957–963.

SCHRUBEN, L. W. AND YÜCESAN, E. 1993. Modeling paradigms for discrete event simulation. *Oper. Res. Lett. 13*, 265–275.

SOM, T. K. AND SARGENT, R. G. 1989. A formal development of event graphs as an aid to structured and efficient simulation programs. *ORSA J. Comput. 1*, 2, 107–125.

YÜCESAN, E. 1989. Simulation graphs: A mathematical framework for the design and analysis of discrete event simulations. Ph.D. Dissertation. Department of Computer Science, Cornell University, Ithaca, NY.

YÜCESAN, E. AND SCHRUBEN, L. 1992. Structural and behavioral equivalence of simulation models. *ACM Trans. Model. Comput. Simul. 2*, 1 (Jan.), 82–103.

Received: December 1998; revised: October 1999; accepted: October 1999