A Discrete Event Model for Real Time System Simulation

Francisco E. Paez, José M. Urriza, Carlos E. Buckle, Lucas Schorb
Facultad de Ingeniería – Depto. de Informática
Universidad Nacional de la Patagonia
Puerto Madryn, Argentina
e-mail: josemuritza@gmail.com

Javier D. Orozco
Depto. de Ingeniería Eléctrica y Computadoras
Universidad Nacional del Sur - CONICET
Bahía Blanca, Argentina
e-mail: jadorozco@gmail.com

Abstract — In this work we present a discrete event model to design and implement a real time system simulator. This kind of software is useful to verify and evaluate algorithms and models, and to compute performance metrics. The discrete event model fits perfectly with discrete dynamical system such as Real Time Systems. The event graph technique is then used as the modeling tool.

Keywords— Real Time Systems simulation; Modeling; Discrete Event Systems.

I. INTRODUCTION

Computer aided simulation is an essential tool in a large number of disciplines. It plays a key role accelerating the creation process of investigation methods and technics. However, the review and validation of simulation software and the techniques used to get these investigation results are often an overlooked issue. It is important that other research groups can validate their results by reproducing experiments using the same simulation software. Or at least with one that uses the same or similar model.

The objective of this work is to formulate a discrete event model for developing Real Time Systems (RTS) simulation software. The model presented was used as basis for our investigation group1 RTS simulation software.

In the past several applications have been developed for RTS simulation: STRESS ([1]), PERTS ([2]), YASA ([3]), Cheddar ([4]), RealTTS ([5]) and the Université Libre de Bruxelles simulator [6], to name a few. Some development modeling tools are MAST ([7]) and FORTISSIMO ([8]). Works that studies simulation tools are MAST ([7]) and FORTISSIMO ([8]).

This paper is organized as follows: Section 2 presents an introduction to RTS, the task model and the notation. Section 3 presents an introduction to the discrete event modeling and simulation, and an overview of the event graph technique. In Section 4 the model development is presented. Section 5 presents a reference implementation. Finally, Section 6 presents our conclusion and future work.

II. REAL TIME SYSTEMS

Stankovic presented in [11] a formal definition accepted by the community discipline: “In real-time computing the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced”.

Depending on how critical it is to meet the deadline, a RTS can be classified as a hard, soft or firm one. A hard RTS does not tolerate any deadline loss. In contrast, a soft RTS can afford to lose some deadlines. Finally a firm RTS typify the losses according to some statistical criterion.

This work uses the single-processor, multiprogrammed system model presented in [12]; the tasks are periodic, preemptable and independent of each other. A scheduling algorithm is used to determine which task has to be executed at a particular instant. This algorithm could perform a static assignation over the shared resource or an assignation based on priorities.

Under this model, a real time task / is characterized by its worst case execution time (Ci), period (Ti) and deadline (Di). A set of n real time tasks is then specified as Γ(n) = (Ci,Di, Ti) . Each task generates an infinite sequence of jobs (instances), where j job of a task τ. The executed time of a job ji at a time t is denoted as ci(t).

Also, in [12] was proved that a single-processor scheduler worst state of load occurs when all jobs require execution simultaneously. This instant is known as critical instant. If all the jobs can execute without missing its deadlines from this instant, then it is said that the RTS is schedulable.

A. Characterization and Analysis of a Real Time System

A dynamical system is one whose state changes in function of time. Then a RTS is such a system. A study of RTS as dynamical systems, when scheduled by Rate Monotonic (RM, [12]) or Deadline Monotonic (DM, [13]) algorithms, can be found in [14]. In this work the RTS evolution since the critical instant is modeled using a fixed point (FP) equation [15] which calculates a task worst response time:

---

1Real Time Systems Group - Universidad Nacional de la Patagonia San Juan Bosco (UNPSJB) Sede Puerto Madryn (http://www.rtsg.unp.edu.ar).
\[ t^{k+1} = C_t + \sum_{j=1}^{n_S} F_j C_j \quad (1) \]

This FP equation models the evolution of the \( \Gamma(i) \) subsystem starting at the critical instant by using an iterative process. The method stops when finds a FP such that \( \bar{t} = t^{k+1} \leq D_p \), in which case the system is schedulable. If the FP is not found after the deadline, the system is not schedulable \((t^{k+1} > D)\). For certain systems such a FP may not exist. The equation \((1)\) is monotonic, deterministic (for each value of \( t \) there exists only one result), non-linear and describes a dynamic system.

Therefore, a RTS could be characterized as a dynamic, non-linear, discrete and deterministic system \((14)\) when it is scheduled by a fixed priority scheduler, such as RM or DM.

III. DISCRETE EVENT SIMULATION

Following there is a brief introduction to discrete event simulation (DES) and the event graph technique is presented.

Discrete event simulation is used to study and simulate systems which can be represented by discrete models\(^2\). In such model, given a finite interval of time, state variables changes instantaneously only in a certain number of moments \((16)\). An event \((v)\) is defined then as an atomic set of changes over the state variables at a certain time.

The models are generally represented as recursive relationships, for example \( t^{k+1} = 2t^k \), where \( k \) denotes the number of discrete steps. Equation \((1)\) is also an example of such model.

During simulations a clock time \( t \) is maintained with the actual simulation time and a future event list \( \Lambda \). This list is a collection of \((t_i, v_i)\) tuples, being \( t_i \geq t \) the instant in which the event \( v_i \) will be executed. Generally \( \Lambda \) is a priority queue prioritized according to the values of \( t_i \).

At each step of the simulation, the first event in \( \Lambda \) (the one with the maximum priority) is executed and \( t \) is updated with the value of \( t_i \). Concurrent processes can be simulated scheduling multiple events at the same time. The simulation starts with a set of events usually scheduled at \( t = 0 \). It ends when one of these situations occurs: \( \Lambda = \emptyset \) (there are no further futures events); \( t \geq t_{end} \) where \( t_{end} \) is a predetermined ending time; or a specific termination event \((v_{end})\) is executed.

A. Event Graphs

A discrete event model can be developed with the Event Graphs technique \((17, 18)\). The dynamics of the modeled system are represented with events which depict system’s state changes. The logical and temporal relationships between them are indicated by edges. It is important to note that the final event graph it is an automaton.

An event graph model \( M \) has the following components:

- \( S \), the set of state variables that conform the system state.
- \( V \), a set of vertices, each corresponding to one event.
- \( E \), the set of directed edges \( e_{ad} = (v_a, e_{ad}, v_b) \) that describes the scheduling relationship between two events \( v_a \) and \( v_b \) in \( V \).

\begingroup
\begin{itemize}
  \item \( F = \{ f_e : S \rightarrow S \forall v \in V \} \), the state changes functions associated with each vertex \( v \in V \).
  \item \( C = \{ e_{ad} : S \rightarrow \{0,1\} \forall e_{ad} \in E \} \), edge condition functions associated with each edge \( e_{ad} \). The edge \( e_{ad} \) is traversed if and only if \( e_{ad} = 1 \).
  \item \( D = \{ \delta_{ad} \in [0,\infty] \forall e_{ad} \in E \} \), the set of time delays. One for each edge \( e_{ad} \).
  \item \( A = \{ A_{e_{ad}} \in E \} \), the set of attributes, if any, associated with each edge \( e_{ad} \).
  \item \( B = \{ B_{v} \in V \} \), the set of parameters, if any, associated with each vertex \( v \).
\end{itemize}

Then an event graph model is specified as the set \( M = (V, E, F, C, D, A, B) \). Each directed edge \( e_{ad} = (v_a, v_b) \) is traversed if and only if the associated edge condition \( e_{ad} \) evaluation is true after the execution of the event \( v_a \). To traverse an edge \( e_{ad} \) means to schedule an event \( v_b \) at the instant \( t + \delta_{ad} \) where \( \delta_{ad} \) is the time delay of the edge \( e_{ad} \). The set of state variables modified by \( f_e \) is known as \( S_a \), \( S_c \subseteq S \).

Given an edge \( e_{ad} \), the associated set of attributes \( A_{e_{ad}} \) will be the formal arguments required by event \( v_b \) (set \( B_{v_b} \)). If no parameters are needed, then \( A \) and \( B \) are empty sets.

It is important to note that \( \Lambda \) and \( t \) (the simulation clock), are associated with a simulation execution of the model. Therefore are not themselves part of \( M \).

The event graph model technique will be used in the next section to model a RTS as a discrete event system.

IV. A DISCRETE EVENT MODEL OF A RTS

Following a discrete event model for a RTS is developed with the event graph technique. The model identifies the instantiation of new jobs, and schedule events for the execution, finalization and preemption of these jobs.

A. System state

The system state \( S \) is composed by a set of \( n \) real-time tasks, \( \Gamma(n) \), and the most recent job \( j_{i,k} \) for each task. The jobs are grouped in a ready queue, sorted according to the task priorities.

B. Events

The model identifies six different events. The first event, \( v_{0} \), corresponds to the RTS setup time. For any job \( j_{i,k} \) the following events are identified: Arrival \((v_{i})\), Execution \((v_{j})\), Finalization \((v_{f})\) and Preemption \((v_{p})\). The \( v_{i} \) event receives the job \( j_{i,k} \) as a parameter. Finally, an event EndSimulation \((v_{es})\) is scheduled at time \( t_{end} \) when the simulation should end.

C. Simultaneous Event Precedence

Two or more events might be scheduled at the same simulated time \( t \). An erroneous execution order of these simultaneous events could result in an invalid state of \( S \). An appropriate execution priority assignment to each event type helps to solve this problem. The priorities assignment is shown in Table 1. The maximum priority is 0.
Listed below are the edge conditions event in case of a $E$.

- $c_{12}$: The previous job of the task $v_i$ has not exceeded its worst case execution time. In that case a new event $v_i$ is scheduled with the task’s next job ($v_{i,k} = 1..n$).
- $c_{13}$: There are no $v_j$ events on $\Lambda$ scheduled for the current simulation time. This means that no new jobs instantiations are scheduled. Then a new $v_j$ event is programmed at the current instant in order to execute the highest priority job at the ready queue.
- $c_{23}$: The highest priority job $j_{i,k}$ could finalize before or at the scheduled time of the nearest $v_j$ event in $\Lambda$. Then it could possibly be preempted by a highest priority job, and a $v_j$ event ($Preemption$) is programmed.
- $c_{24}$: The highest priority job $j_{i,k}$ could not finalize before or at the scheduled time of the nearest $v_j$ event in $\Lambda$. Then a new $v_j$ event is programmed at the current instant in order to execute it.
- $c_{32}$: There are no $v_j$ events on $\Lambda$ for the current simulation time and there is at least one job $j_{i,k}$ at the ready queue. Then a new $v_j$ event is programmed at the current instant in order to execute it.
- $c_{34}$: The edge condition $c_{32}$ avoids the duplication of a $v_j$ event in case of a $v_j$ event is scheduled at the same time. Notice that the edge conditions $c_{23}$ and $c_{24}$ are mutually exclusive.

### Table I

<table>
<thead>
<tr>
<th>Event</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization ($v_0$)</td>
<td>0</td>
</tr>
<tr>
<td>EndSimulation ($v_i$)</td>
<td>1</td>
</tr>
<tr>
<td>Finalization ($v_j$)</td>
<td>2</td>
</tr>
<tr>
<td>$Preemption$ ($v_k$)</td>
<td>3</td>
</tr>
<tr>
<td>$Arrival$ ($v_l$)</td>
<td>4</td>
</tr>
<tr>
<td>$Execution$ ($v_m$)</td>
<td>5</td>
</tr>
</tbody>
</table>

#### D. Edges

The events $v_0$ through $v_4$ are connected by six edges, as shown in Figure 1. The edges are:

- $e_{01}$: Schedule the arrival of the first job of each task ($j_{i,1}$).
- $e_{11}$: Schedule the next event $v_i$ ($Arrival$).
- $e_{12}$: Schedule a new event $v_j$ ($Execution$).
- $e_{23}$: Schedule a new event $v_j$ ($Finalization$).
- $e_{24}$: Schedule a new event $v_k$ ($Preemption$).
- $e_{34}$: Schedule a new event $v_m$ ($Execution$).

#### E. Edge condition functions

The edge $e_{01}$ schedules the first job of each task. Then it is traversed only at the simulation’s start. In this work it hasn’t got an associated edge condition, but one could be added to, for example, perform schedulability analysis test. Listed below are the edge conditions $e_{01}$:

- $c_{12}$: The previous job of the task $v_i$ has not exceeded its worst case execution time. In that case a new event $v_i$ is scheduled with the task’s next job ($v_{i,k}$).
- $c_{13}$: There are no $v_j$ events on $\Lambda$ scheduled for the current simulation time. This means that no new jobs instantiations are scheduled. Then a new $v_j$ event is programmed at the current instant in order to execute the highest priority job at the ready queue.
- $c_{23}$: The highest priority job $j_{i,k}$ could finalize before or at the scheduled time of the nearest $v_j$ event in $\Lambda$. Then it could possibly be preempted by a highest priority job, and a $v_j$ event ($Preemption$) is programmed.
- $c_{24}$: The highest priority job $j_{i,k}$ could not finalize before or at the scheduled time of the nearest $v_j$ event in $\Lambda$. Then a new $v_j$ event is programmed at the current instant in order to execute it.
- $c_{32}$: There are no $v_j$ events on $\Lambda$ for the current simulation time and there is at least one job $j_{i,k}$ at the ready queue. Then a new $v_j$ event is programmed at the current instant in order to execute it.
- $c_{34}$: The edge condition $c_{32}$ avoids the duplication of a $v_j$ event in case of a $v_j$ event is scheduled at the same time. Notice that the edge conditions $c_{23}$ and $c_{24}$ are mutually exclusive.

#### F. Time delays

A $v_j$ event should be scheduled for an instant $t_j \geq t$. In the event graph model this is expressed by associating a time delay $\delta_{ab} \geq 0$ to each edge $e_{ab}$ such that $t_j = t + \delta_{ab}$.

The edge $e_{01}$ schedules a new event $v_i$, which represents a new job of a task $i$. The time delay $\delta_{i}$ is calculated with:

$$\delta_i = \frac{t + T_i - T_i}{T_i - t}$$

The event $v_2$ is scheduled at the current time by the edges $e_{12}$ and $e_{24}$. Then $\delta_{23} = \delta_{24} = 0$. The time delay for the edge $e_{23}$ is $\delta_{23} = C_i - c_i(t)$, which is the remnant execution time of the job $j_{i,k}$.

The edge $e_{24}$ is traversed when the nearest $v_j$ event in $\Lambda$ is scheduled for a time $t_j < t + C_i$. Then a $v_j$ event is scheduled at $t_j$ instant. So $\delta_{24} = t_j - t$.

#### G. Event execution

In this section we present the modifications to the system state ($S$) that each event performs when executes. These are the functions $f_j$ for each $v_j$ event.

The $v_j$ event ($Initialization$) schedules the task’s initial jobs. So the first $v_j$ event of each task is programmed from $v_0$. The simulation clock is also initialized, generally with $t = 0$. Any other activity that should be done at setup time (i.e. worst case response time analysis) is performed at this event.

The $v_j$ event ($Arrival$) adds a new job $j_{i,k}$ to the scheduler ready queue. Then it schedules a new $v_j$ event for the next task job $j_{i,k+1}$ adding ($t + \delta_{13}, v_{j(i,k+1)}$) into $\Lambda$. Finally if the edge condition $c_{23}$ is met, a $v_j$ event ($Execution$) is scheduled for that instant, $\Lambda \leftarrow (t, v_j)$.

In order to perform the simulated execution of the highest priority job at the ready queue, the method or routine that implements the simulated scheduler logic should be invoked at the $v_j$ event execution. If the edge condition $c_{23}$ is satisfied, then a $v_j$ event ($Finalization$) is programmed ($\Lambda \leftarrow (t + \delta_{23}, v_j)$). Otherwise, a $v_j$ event ($Preemption$) is scheduled, $\Lambda \leftarrow (t + \delta_{24}, v_j)$.
Finally the \( v_1 \) event (EndSimulation) is scheduled at the instant \( t_{\text{final}} \) where the simulation should end. It must free any resources and invoke the auxiliary routines, like report generation.

### H. Model execution example

An execution example of the model is shown at next; For this we will use the RTS \( T(3) = \{ (1, 3, 3), (1, 4, 4), (1, 6, 6) \} \). The next table shows the model evolution until \( t = 6 \).

**Table II**

<table>
<thead>
<tr>
<th>( t )</th>
<th>Event</th>
<th>( \Lambda )</th>
<th>Executed job</th>
<th>Ready queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ( v_0 )</td>
<td>(0, ( v_1(j_{1,0}) )), (0, ( v_2(j_{1,0}) ))</td>
<td>( - )</td>
<td>( - )</td>
<td></td>
</tr>
<tr>
<td>0 ( v_1(j_{1,0}) )</td>
<td>(0, ( v_1(j_{1,0}) )), (0, ( v_2(j_{1,0}) ))</td>
<td>( - )</td>
<td>( j_{1,0} )</td>
<td></td>
</tr>
<tr>
<td>0 ( v_2(j_{1,0}) )</td>
<td>(0, ( v_1(j_{1,0}) )), (3, ( v_1(j_{1,1}) )), (4, ( v_2(j_{1,0}) ))</td>
<td>( - )</td>
<td>( j_{1,0}, j_{1,0} )</td>
<td></td>
</tr>
<tr>
<td>0 ( v_3(j_{1,0}) )</td>
<td>(0, ( v_2(j_{1,0}) )), (3, ( v_1(j_{1,1}) )), (6, ( v_2(j_{1,0}) ))</td>
<td>( - )</td>
<td>( j_{1,0}, j_{1,0}, j_{1,0} )</td>
<td></td>
</tr>
<tr>
<td>0 ( v_2 )</td>
<td>(1, ( v_3(j_{1,0}) )), (4, ( v_1(j_{1,1}) )), (6, ( v_2(j_{1,0}) ))</td>
<td>( j_{1,0} )</td>
<td>( j_{2,0}, j_{1,0} )</td>
<td></td>
</tr>
<tr>
<td>1 ( v_3 )</td>
<td>(1, ( v_3(j_{1,0}) )), (4, ( v_1(j_{1,1}) )), (6, ( v_2(j_{1,0}) ))</td>
<td>( j_{1,0} )</td>
<td>( j_{2,0}, j_{1,0} )</td>
<td></td>
</tr>
<tr>
<td>1 ( v_2 )</td>
<td>(2, ( v_3(j_{1,0}) )), (3, ( v_1(j_{1,1}) )), (4, ( v_2(j_{1,0}) ))</td>
<td>( j_{1,0} )</td>
<td>( j_{1,0} )</td>
<td></td>
</tr>
<tr>
<td>2 ( v_3 )</td>
<td>(2, ( v_3(j_{1,0}) )), (3, ( v_1(j_{1,1}) )), (4, ( v_1(j_{2,1}) )), (6, ( v_2(j_{1,0}) ))</td>
<td>( j_{1,0} )</td>
<td>( j_{1,0} )</td>
<td></td>
</tr>
<tr>
<td>2 ( v_2 )</td>
<td>(3, ( v_3(j_{1,0}) )), (3, ( v_1(j_{1,1}) )), (4, ( v_2(j_{1,0}) )), (6, ( v_2(j_{2,1}) ))</td>
<td>( j_{1,0} )</td>
<td>( - )</td>
<td></td>
</tr>
<tr>
<td>3 ( v_3 )</td>
<td>(3, ( v_3(j_{1,0}) )), (4, ( v_2(j_{1,0}) )), (6, ( v_2(j_{2,1}) ))</td>
<td>( j_{1,0} )</td>
<td>( - )</td>
<td></td>
</tr>
<tr>
<td>3 ( v_2(j_{1,0}) )</td>
<td>(3, ( v_3(j_{1,0}) )), (4, ( v_2(j_{1,0}) )), (6, ( v_2(j_{2,1}) ))</td>
<td>( - )</td>
<td>( j_{1,1} )</td>
<td></td>
</tr>
<tr>
<td>3 ( v_2 )</td>
<td>(4, ( v_3(j_{1,0}) )), (4, ( v_2(j_{1,0}) )), (6, ( v_2(j_{2,1}) ))</td>
<td>( j_{1,1} )</td>
<td>( - )</td>
<td></td>
</tr>
<tr>
<td>4 ( v_3 )</td>
<td>(4, ( v_3(j_{1,0}) )), (6, ( v_2(j_{2,1}) ))</td>
<td>( j_{1,1} )</td>
<td>( - )</td>
<td></td>
</tr>
<tr>
<td>4 ( v_2(j_{2,1}) )</td>
<td>(4, ( v_3(j_{1,0}) )), (6, ( v_2(j_{2,1}) )), (8, ( v_2(j_{2,2}) ))</td>
<td>( - )</td>
<td>( j_{1,1} )</td>
<td></td>
</tr>
<tr>
<td>4 ( v_2 )</td>
<td>(5, ( v_3(j_{1,0}) )), (6, ( v_2(j_{2,1}) ))</td>
<td>( j_{1,1} )</td>
<td>( - )</td>
<td></td>
</tr>
<tr>
<td>5 ( v_3 )</td>
<td>(6, ( v_3(j_{1,0}) )), (6, ( v_2(j_{2,1}) )), (8, ( v_2(j_{2,2}) ))</td>
<td>( j_{1,1} )</td>
<td>( - )</td>
<td></td>
</tr>
<tr>
<td>6 ( v_1(j_{1,1}) )</td>
<td>(6, ( v_2(j_{2,1}) )), (8, ( v_2(j_{2,2}) ))</td>
<td>( - )</td>
<td>( j_{1,1} )</td>
<td></td>
</tr>
</tbody>
</table>

At the instant \( t = 0 \) the future event list \( \Lambda \) contains the initials \( v_1 \) events, one for each task’s first job. These events are generated by \( v_0 \). Then, the first \( v_1 \) event in \( \Lambda \) is executed, \( v_1(j_{1,0}) \). It schedules a new \( v_1 \) event at \( t = 3 \) for the next job of \( r_1 \) (which is \( j_{1,0} \)). This implies to put \( (3, v_1(j_{1,0})) \) into \( \Lambda \). The execution of the events \( v_1(j_{2,0}) \) and \( v_1(j_{1,0}) \) is analogous. When the last \( v_1 \) event on \( \Lambda \) is executed, the edge condition \( c_{2,3} \) is then valid. Therefore, a \( v_2 \) event is scheduled at \( t = 0 \). This event will simulate the execution of the highest priority job in the scheduler ready queue \( j_{1,0} \). As the job can finish before any future event on \( \Lambda \), the edge condition \( c_{2,3} \) is satisfied. Then a \( v_3 \) event is schedule at \( t = 1 \). With no more events of any kind in \( \Lambda \) for \( t = 0 \), the simulation clock is then advanced to the instant time at which the next event is scheduled \( t = 1 \). Then a new \( v_2 \) event is scheduled at \( t = 1 \) in order to continue the simulation of the jobs in the ready queue. The rest of the model execution presents an analogous behavior.

### V. Implementation

Next a reference implementation in Java using the SSJ ([19]) simulation library is described. This library offers the package `simmEvents` for discrete event simulation, which has two main classes, Simulator and Event.

The class Simulator represents the simulation executive. It provides the simulation clock and the future event list \( \Lambda \) (offering multiple implementations). Also provides methods to start and stop the simulation.

The Event class represents an abstraction of an event. Each one of the events presented in section 4B should be implemented as a class which extends it (Init, Arrival, Run, End and Preempt). These classes must override the action(s) method in order to perform the corresponding actions (that is the \( f_i \) functions listed in section 4G).

It is assumed that a collection or array with the RTS to simulate is provided. Also there must be auxiliary classes that implement the scheduler and other techniques to be evaluated.

The Init class \((v_{0}\) event) creates the initial instances of the Arrival class. These instances are scheduled at the appropriate times using the schedule(d)elay method. The Arrival class adds its associated job into the ready queue using the actions() method. If the condition \( c_{1,2} \) is met, it should schedule a new instance of Run \((v_{2}\) class at the current time. This class should invoke any method required to perform modifications at \( S \). If the edge conditions \( c_{2,3} \) or \( c_{3,4} \) are valid it should schedule an instance of End \((v_{3}\) or Preempt \((v_{4}\) class) should schedule an instance of Scheduler.finishTask() method while Preempt will invoke Scheduler.preemptTasks(). This way the scheduler logic is decoupled from the event logic. If the edge condition \( c_{3,4} \) is met, End class must schedule a new Run class instance for the current time. The Simulator class from SSJ provides methods that help to verify the different edge conditions.

### VI. Conclusions and Future Work

In this paper we have presented a discrete event model for the simulation of a RTS, based on the analysis of a RTS as dynamic, discrete, non-linear and deterministic system. The event graph technique was used as modeling tool due to its simplicity and ease of implementation. This model brings a framework in which simulation software can be developed using one of the many DES packages or libraries available. Also, this work serves as basis for future developments that extends the presented model; for example, for simulating heterogeneous RTS.

### VII. References


