

Automata driven approach to simulation of Embedded Computing Systems.

Carl W. Entemann, Alexander B. Wolpert
School of Computer Science and Telecommunications
Roosevelt University, 430 S. Michigan Ave.,
Chicago, IL 6606605, USA.

Abstract. The goal of this paper is to develop a new formalism for modeling of embedded systems. We introduce a new formalism based on timed automata. Subsequently, we show that it is strictly more expressive than closed queuing networks with constant times that are often used in simulation. We introduce conversion algorithm and estimate its complexity. We formulate a simulation method that work in real-time and is faster than traditional closed queuing network simulation.

Keywords. Discrete event simulation, Scheduling, Embedded Systems, Timed Automata.

1. Introduction

In embedded systems design it is often necessary to evaluate throughput and logical correctness of an event sequence occurring within an execution run of a system under design. An embedded system can be regarded as a set of resources (possibly changing in time), a set of processes, and a set of resource-controlling algorithms. A process could be thought of as an entity that moves between possible states. In every state the set of possible successor states is well known, in each state a process spend certain amount of time and then instantaneously move to one of its successor states. To proceed at any stage of its evolution a process requires resources. The resource can be given to the process by an algorithm that controls the resource distribution. The availability of required resources uniquely defines next state of the process.

When times that are spend by the process in each state and internal resources do not vary, processes can be represented by a Petri nets. Yet the parameters of composite system that consist of a number of processes may vary because of interaction between processes. In this case 'pure' automata or Petri nets may not be an appropriate tool for simulation and modeling. Tools for simulation of such systems were developed within the Petri net framework (see [1]). Elsewhere [2] we showed that these tools are indeed adequate for simulation of certain embedded systems. However, Petri net modeling of complex systems may result in incomprehensible nets. In many cases automata-theoretic approach may help. Tools somewhat similar to [1] were developed in automata-theoretic framework (see [3]), but they are not quite adequate for simulation of embedded systems (see section 5). We extend the methods of [3] in order to design adequate simulation language for modeling multi-process embedded systems.

In particular one of the problems is that scheduling algorithms are defined fro a single resource only. That creates a problem when a system state is defined. This problem was considered in [4] and [5]. However, the methods developed in previous research do not account for preemptive resource distribution algorithms which are the primary object of interest in this study.

2. Organization of the paper.

Consider the following example (*PLC-robot*):

Programmable Logic Controller (PLC) controls distance sensor and angle sensor of the robot. Based on these measurement a processing program works out controls executed by a robot. Sensors are neither synchronized, nor can they exchange information bypassing PLC. To explain system evolution let us imagine that PLC controls just one sensor. The process evolution in this case can be described as follows: PLC starts the process of measurement by turning the sensor on. When the data is collected, the sensor signals to PLC the end of current measurement operation and goes to sleep. PLC processes acquired data and forms a control sequence to the executive element. Upon the completion of the this step sensor is turned on and is restarted again. Processes associated with both sensors are identical. In this system one process may signal to PLC the end of measurement step during the processing step of another process . The former is either allowed to interrupt the processing phase of the latter or it may be forced to wait until the end of the information processing step of the second sensor process. In this example we assume that distance sensor may interrupt angle sensor because its data is changing faster and is more important. Obviously, this system adheres to assumptions made in section 1.

One may formally represent the system as two cyclic processes, each consisting of two repeatedly executed steps:

1. Data acquisition;
2. Data processing and formulation of control influence.

PLC is a shared resource since steps two of the processes cannot be executed simultaneously. The resource distribution algorithm is Priority scheduling since distance measurements are considered more important than angle measurements.

We first address an issue of modeling time-variable systems. Traditionally, these systems (like the one described in this

section) are modeled by closed queuing networks [6]. However, queuing theory developed elaborate tools for statistical analysis of system behavior rather than tools for deterministic modeling of individual that we are interested in. To extend queuing systems language to modeling of individual behavior one need to expose implicit assumptions routinely presumed to be true in queuing. Therefore, in the section 3 we expose these assumptions and introduce the language of systems description that explicitly incorporates them. This language is used in the section 3 for model-definition of our example. In section 4 we construct an algorithm that transforms a system description in the language of closed queuing networks into automata theoretic language. In section 5 we attempt to use timed automata to describe our examples. We prove that expressive strength of timed automata is incomparable to expressive strength of closed queuing networks with constant times We modify the definition of timed automata to be able to express closed queuing networks in it. Using intricacies of new description formalism we formulate a simulation method that work in real-time and is faster than traditional closed queuing network simulation In section 6 we discuss our results.

3. Language of Model Definitions.

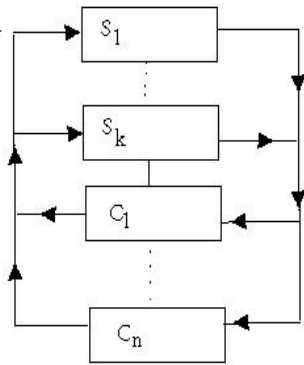


Fig. 1.

In closed queuing networks a part of a system description is a strongly connected directed graph $G=(V,E)$ that is called a closed queuing system graph. Here V is a set of nodes of cardinality $|V|$ and E is a set of arcs (see fig 1.). A closed queuing network serves m calls from the set $M=\{1,2,\dots,m\}$. Each call is served by executing a sequence of steps, that forms a loop, i.e.:

- Each step of a call is mapped into the node of queuing system so that if step k is mapped to node n_k and the step $k+1$ is mapped into the node n_{k+1} then (n_k, n_{k+1}) belongs to a set of arcs of G . Here we assume that every step can be executed in exactly one node of the network. Therefore, each step l_i is mapped to a node of the graph V where this step can be executed, $m_i: \{0,1,\dots,s_i\} \rightarrow V$.
- The number of steps, their sequence in terms of following from one node to another in the queuing network, and all other characteristics of steps such as duration, required resource utilization, etc., are determined only by call's number, e.g., i^{th} call is served by executing steps $0,1,\dots,s_i$ and then find itself again in the step 0 and in the second run of service.

This loop continues indefinitely. The discrete state of i^{th} call is a number of the step l_i that is currently executed by a call.

Lets say that a calls behavior is autonomous if no interference from other calls is encountered. To define autonomous behavior of i^{th} call it is enough to define current discrete state of the call and an infinite sequence $\tau_{i_0}, \tau_{i_1}, \tau_{i_2}, \dots$ where τ_{i_0} is the time left of the current step l_i , and τ_{i_u} is a duration of the step $[l_i+u]_{\text{mod } S_{i+1}}$ in the run number $k+\lfloor u/s_i \rfloor$. if k is the number of current run. The state of the call is a pair:

<the discrete state of the call, the sequence $\{\tau_{i_j}\}_{j=0}^{\infty}$.

The autonomous evolution of i^{th} call can be described as follows:

- For $0 \leq t < \tau_{i_0}$ the state of the call at the time t has the same discrete component l_i and $\tau_{i_0}^t, \tau_{i_1}, \dots, \tau_{i_j}, \dots$, where $\tau_{i_0}^t = \tau_{i_0} - t$ and other components of the sequence are the same as they where at the time 0;
- For $\tau_{i_0} \leq t < \tau_{i_0} + \tau_{i_1}$ the state of the call at the time t has the discrete component $[l_i+1]_{\text{mod } S_{i+1}}$ and $\tau_{i_0}^t + \tau_{i_1} - t, \tau_{i_2}, \dots$;
- for $\sum_{j=0}^n \tau_{i_j} \leq t < (\sum_{j=0}^n \tau_{i_j}) + \tau_{i_{n+1}}$ the state of the call at the time t has the discrete component $[l_i+n+1]_{\text{mod } S_{i+1}}$ and continuous component $(\sum_{j=0}^n \tau_{i_j}) + \tau_{i_{n+1}} - t, \tau_{i_{n+2}}, \dots$.

Apparently, in an embedded system more than one call coexist at any given time. Their cooperation and competition must be reflected in the model. Therefore, the system description is not barely a combination of autonomous states of state of all calls.

We now proceed to the description of the state of the system as a whole. The discrete state of the system is given by a set of queues, one for each node of the graph G . The queue at a node S denoted as Q_S is an ordered set (possibly empty) of pairs $Q_S = [(i_1, l_{i_1}), (i_2, l_{i_2}), \dots, (i_m, l_{i_m})]$, where the first component of a pair is a call number and the second component is the discrete state of a call. A call with its discrete state is a member of exactly one queue in the system. ($Q_S \cap Q_j = \emptyset$) and this membership is consistent with the mapping of a current step to a node of the graph G , namely $(i_j, l_{i_j}) \in Q_S \Rightarrow m_{i_j}(l_{i_j}) = S$. It is assumed that the first pair in the list Q_S is being served by a node S . Therefore, the discrete state space Θ of a system is the set of all consistent combinations of queues Q_S at all nodes $S \in V$. Note that the calls in the system may interfere with each other because of competition for the resources. Presumably, the interference between calls may not change the step sequence or a step duration, but it may cause pauses in the service of the call (step execution). A pause may take place between the steps of a call (pause in transition between discrete states of a call) and it may be found within the currently executing step of a call (discrete component of the state of a call.) This fact is reflected in the current state of the system that is a combination of discrete state of the system and infinite sequences:

$$\{\tau_{1j}\}_{j=0}^{\infty}, \{\tau_{2j}\}_{j=0}^{\infty}, \dots, \{\tau_{mj}\}_{j=0}^{\infty}$$

where i^{th} line is a sequence of duration of steps of i^{th} call enumerated mod s_i and τ_{i_0} is the duration of the rest of a current step.

The fact that there could be a pause between the execution of the parts of the steps is reflected in the position of i^{th} call in the queue of the node S , namely if a pair (i, τ_{i_0}) is not the first one in the queue of the node S and $\tau_{i_0} \neq \tau_{i_1}^1$, where $\tau_{i_1}^1$ is the value of the second member of the sequence at a time of a previous discrete state of the call $([l_i - 1]_{\text{mod}(s_i+1)})$ then apparently, there is a pause in service of the step l_i of i^{th} call by the node S . This pause takes place between the intervals $[0, \tau_{i_1}^1 - \tau_{i_0})$ and $[\tau_{i_0}]$ of current step l_i .

Formally resource is a graph node. Algorithms governing resources' distribution also known as scheduling disciplines are represented in the transition rules of discrete state of a system and ultimately in the systems evolution. Let DS be a subset of a M , Θ , $DS = \{(i, [Q_1, Q_2, \dots, Q_{|V|}]) \mid \exists j \in [1, \dots, |V|] ((i, l_j) = F(Q_j))\} \in \mathcal{S}M, \Theta$. such that a call in the left position of a pair at its current state is served by one of the nodes, i.e., $(i, l_j) = F(Q_j)$ denotes first in the list Q_j . One can define a global scheduling algorithm for a system as a map $\text{Sched}: DS \hat{\Theta} :: (i, [Q_1, Q_2, \dots, Q_{|V|}]) \rightarrow [Q^1_1, Q^1_2, \dots, Q^1_{|V|}]$ such that it is consistent with a step sequence of i^{th} call, i.e. if $(i, l_j) = F(Q_j)$ before the application of Shed then $\text{Shed}(i, [Q_1, Q_2, \dots, Q_{|V|}])$ is such that $(i, [l_i + 1]_{\text{mod}(s_i+1)}) \in Q^1_k$ where k is the resource to which $[l_i + 1]_{\text{mod}(s_i+1)}$ is mapped. In effect the consistency restrictions imposed on Shed come from the domain interpretation of the scheduling algorithm: it must take i^{th} call that just completed its step l_i to a node where next step $[l_i + 1]_{\text{mod}(s_i+1)}$ is going to be executed. Queuing theory is interested in so called local scheduling algorithms that depend only on the numbers of the calls in queue, their current steps and the the order of arrival in queue. These types of scheduling algorithms being independent from the queues of all other nodes allow simple algorithmic formulation as sorting algorithms. Moreover, it allows to reconstruct a Sched algorithm as a combination of partial algorithms defined on a far less complex state spaces.

Let the state spaces DS_k of a node K be the set of all possible queues at this node. Then a partial resource scheduling algorithm for the node K is a defined as a mapping $\text{P}_{S_k}: DS_k \hat{\Theta} DS_k$. Obviously, if such mappings are defined for each node then corresponding Sched mapping can be constructed¹.

Evolution of the system can be described as follows:

$$\text{Let} \quad \tau = \min_{i \in M} \tau_{i_0} \quad (3.1)$$

and let $J\mathcal{S}M$ be the set of indices on which the minimum is achieved. Then for $0 \leq t < \tau$ the evolution is described by subtracting t from τ_{i_0} for all j such that $(j, l_j) = F(Q_k)$ for a $k \in J \mid V$. Note that the discrete component of the system state stays the same during this time. At the time $t = \tau$ the system switch to a new discrete component of the state. In vast majority of models where the minimum must be achieved on some unique i_0 ² the new discrete component of the system state is determined as $\text{Sched}((i_0, [Q_1, Q_2, \dots, Q_{|V|}])) = [Q^1_1, Q^1_2, \dots, Q^1_{|V|}]$. For some models may be necessary to allow $|J| > 1$. This assumption means that limited number of events can happen at the same instant of time in an underlying computer system. However, in real systems usually an order that is inherent for a system is imposed on a set of simultaneous events. For example it is conceivable that both sensors of PLC robot finish data acquisition phase simultaneously. Yet one of the two will be closer to the arbitrator on an I/O bus and therefore the event generated by this sensor will be treated as the one that happened first. Here we assume that such ordering algorithm $\text{Ord}: \{1, 2, \dots, |J|\} \hat{\Theta} J$ always exist (if nothing else then in the way we pick one index out of J .) Therefore, the evolution here is no different from the case when $|J| = 1$ with two exceptions:

- Sched is applied to the first in order event from the set of events that happen at the same instance of time $\text{Sched}((i_0, [Q_1, Q_2, \dots, Q_{|V|}])) = [Q^1_1, Q^1_2, \dots, Q^1_{|V|}]$ where $i_0 = \text{Ord}_j(1)$;
- It becomes possible that $\tau = \min_{i \in M} \tau_{i_0} = 0$ and therefore it is possible that a series of changes of discrete component of state may happen before next continuous evolution. Apparently, the number of such changes is limited by finiteness of J .

We now get back for a moment to the system state definition. One basic assumption for the model definition was that for each call duration of all steps in future are well known. This assumption is not quite realistic and due to it the model has infinite number of parameters. To be able to use the model of the system for analysis this assumption must be substituted by a new notion that is consistent with the previous model definition but reduces the number of model parameters to a finite value. One way to achieve this goal is to introduce functional parameters that will associate to a call number and a step number its duration. This is the way chosen by queuing theory. It assumes that each step duration is a random variable distributed according to some distribution that depends only on the call number i and the step number in the cyclic step enumeration mod q_i . This concludes the model definition in the language of queuing networks.

Yet few more words have to be said about queuing networks before we proceed to reformulation of this model. The

¹ Note that though assumption of local resource distribution algorithm is true in most cases, there still exist systems that may be modeled as queuing systems with global resource distribution algorithms.

² Cardinality of the set J is 1 ($J = \{i_0\}$) stands for what is known as Markov property of a random process when no two events can happen at the same instant of time [6]

distributions that are almost universally used in these models are exponential distributions the reason being that they are suitable for analytical solution of the problems associated with models. Any model in the introduced language with the assumption of exponential duration of steps will become a Markov chain with continuous time defined on the finite state space of discrete components of the system. [6]. However, often this assumption is not realistic enough. In many cases (see our robot example) the duration of each step is always 'almost' the same with very little variation. Then either deterministic distribution or normal distribution of steps should be chosen for an honest model. Unfortunately, it is very difficult (if possible) to analyze models resulting from these assumptions by means of queuing networks.

The construction outlined in the language of queuing theory resembles the constructions of some timed automata. Reformulation of the aforementioned construction in the automata-based language is the goal of this paper.

Lets make sure that queuing networks language is indeed adequate for definition of embedded systems in general and for our examples in particular. For the examples 1 and 2 of the section 2 the queuing network graph is given on Fig. 1. Consider example 1, then $V=\{PLC, S1, S2\}$ where S1 and S2 stand for sensor 1 and sensor 2 correspondingly, $E=\{(S1,PLC), (PLC,S1), (S2,PLC), (PLC,S2)\}$. The autonomous state of the processes $M=\{1,2\}$ is given by steps $S_1=S_2=\{0,1\}$ with the mapping of steps to nodes $N_i:M \Rightarrow V::0 \rightarrow S_i::1 \rightarrow PLC$, where $i \in \{1,2\}$.

Assume deterministic distribution of duration of steps $Time:M * S \Rightarrow R^{|M * S|}::(1,0) \rightarrow T^0_1::(1,1) \rightarrow T^1_1::(2,0) \rightarrow T^0_2::(2,1) \rightarrow T^1_2$ where $T^j_i \in R$. The discrete state space of the system and the scheduling function is defined as follows:

The scheduling function is defined as follows:

Sched: $M * \Theta \Rightarrow \Theta::(1, \{\emptyset; [(1,0)]; [(2,0)]\}) \rightarrow \{(1,1); \emptyset; [(2,0)]\}$
 $::(2, \{\emptyset; [(1,0)]; [(2,0)]\}) \rightarrow \{(2,1); [(1,0)]; \emptyset\}$
 $::(1, \{(1,1); \emptyset; [(2,0)]\}) \rightarrow \{\emptyset; [(1,0)]; [(2,0)]\}$
 $::(2, \{(1,1); \emptyset; [(2,0)]\}) \rightarrow \{(1,1)(2,1); \emptyset; \emptyset\}$
 $::(2, \{(2,1); [(1,0)]; \emptyset\}) \rightarrow \{\emptyset; [(1,0)]; [(2,0)]\}$
 $::(1, \{(2,1); [(1,0)]; \emptyset\}) \rightarrow \{(1,1)(2,1); \emptyset; \emptyset\}$
 $::(1, \{(1,1)(2,1); \emptyset; \emptyset\}) \rightarrow \{(1,1); [(1,0)]; \emptyset\}$

	Q_{PLC}	Q_{S1}	Q_{S2}	#
$\Theta =$	$\{\emptyset;$	$[(1,0)]$	$[(2,0)]\}$	1
	$\{[(1,1);$	$\emptyset;$	$[(2,0)]\}$	2
	$\{[(2,1);$	$[(1,0)]$	$\emptyset\}$	3
	$\{[(1,1),(2,1);$	$\emptyset;$	$\emptyset\}$	4
	$\{[(2,1),(1,1);$	$\emptyset;$	$\emptyset\}$	5

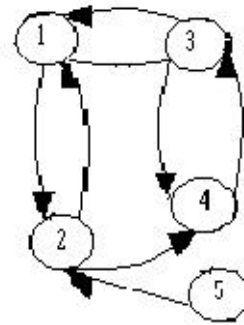


Fig 2

Because of the higher priority of the S1 over S2 that the state $\{(2,1),(1,1); \emptyset; \emptyset\}$ is never reached. Assuming we are interested in a steady state behavior, unreachable states can be removed from system definition

4. Automata Hidden in the Simulation of Closed Queuing Networks.

When applying discrete-event simulation techniques to simulation of a closed queuing network one is supposed to emulate the queuing network operations in the iterative step of a simulation algorithm (see for example [de simulation textbook]). The following algorithm is an implementation of aforementioned approach:

Closed Queuing Network Simulation (CQNS) Algorithm.

Input:

- For each call $i \in \{1, \dots, m\}$, the map of call steps into nodes of the graph of closed queuing network. The map is supplied in the format (i, j, v) where i is a call number, j is a step number, and v is a node number of a node in the queuing network graph;
- For each $i \in \{1, \dots, |V|\}$, the local scheduling algorithm. Each scheduling algorithm has the input of i^{th} -queue state and the output of a new i^{th} -queue state and a pair (i, l_i) extracted from the i^{th} -queue
- For each pair (i, l_i) , the time generating algorithm $TimeDist(i, l_i)$;
- For all $i \in \{1, \dots, m\}$, initial steps (i, l_i) ;
- For each $i \in \{1, \dots, |V|\}$, initial queue states Q_i ;
- Global Time of Simulation (GTS).

Output: Event sequence with time stamps.

Initialization:

1. For all $i \in \{1, \dots, m\}$ set $\text{ResT}(i, l_i) = \text{TimeDist}(i, l_i)$; <Set times of initial steps>
2. $\text{CurrentTime} = 0$ <Set global time to 0>

Iteration:

1. $\text{Comp} = \emptyset$; <Initialize the list of potentially completed (call, step) pairs to empty list>
2. for every j ranging from 1 to $|V|$ <Create a list of pairs (call, step) that can be potentially completed in this system state>
 - do
 - 2.1. apply the local scheduling algorithm to the queue Q_j to find a (i, l_i) call-step pair that has the resource;
 - 2.2. $\text{Comp} = \text{Append}(\text{Comp}, (i, l_i))$
 - od
3. Find T_{\min} minimum among residual times of all $(i, l_i) \in \text{Comp}$; <determine time of the next event, i.e., find minimum time needed to complete at least one pair on the list of those that can be completed in this network state>
4. Decrease residual times by T_{\min} for all $(i, l_i) \in \text{Comp}$,
 $\text{ResT}(i, l_i) = \text{ResT}(i, l_i) - T_{\min}$;
 $\text{CurrentTime} = \text{CurrentTime} + T_{\min}$; <Decrease residual time of all pairs that can be completed in this network state by the time that passes until the next event. Increase current global time>
3. Find first (i, l_i) in Comp such that residual time $\text{ResT}(i, l_i) = 0$ and remove it from its queue, $\text{REMOVE}((i, l_i), Q_j)$; <Choose next event among those that happen simultaneously>
4. Output (i, l_i) and CurrentTime ;
4. $\text{ResT}(i, [l_i+1]_{\text{mod}(S_i+1)}) = \text{TimeDist}(i, [l_i+1]_{\text{mod}(S_i+1)})$; <Generate a new residual time for the next step of the call that completed its step according to the distribution associated with the step>
5. $Q_{m_j(i, [l_i+1]_{\text{mod}(S_i+1)})} = \text{INSERT}((i, [l_i+1]_{\text{mod}(S_i+1)}), Q_{m_j(i, [l_i+1]_{\text{mod}(S_i+1)})})$; <Insert the next step of the call that completed a step into appropriate queue>
6. If $\text{CurrentTime} > \text{GTS}$ STOP else go to Iteration

In the section 3 we agreed to consider only those local scheduling algorithms that select a pair (call, step) to which the resource is assigned based only on pairs that are in a queue and on their order in the queue. Therefore, the scheduling algorithm computes the scheduling criteria (key) for each pair and sorts pairs to find the maximal (least) element with respect to the key value. It follows that the key plays a role of priority, and the simulation algorithm in its iteration step is dealing with sorting multiple priority queues. Depending on the data structure used to implement priority queues, a scheduling algorithm (considered with its INSERT and REMOVE parts) runs no faster than $O(\log N)$. Here N is a number of pairs in the queue that in our case is limited by the number of calls m . Obviously, if we could eliminate steps 2, 5, and 7 it would substantially improve the algorithm.

We intend to substitute CQNS algorithm by a new algorithm that generates a run of a deterministic automaton as an output and we intend to prove that CONS generate the same output. The new algorithm will have steps analogous to Algorithm 1 except steps 2, 5, and 7 that are executed in constant time.

To construct a desired automaton we define its states as corresponding discrete states of a closed queuing system. The alphabet of the automaton is a set of all legitimate pairs (call, step). A triplet $(S, (i, l_i), T)$ is a transition of the automaton if T can be received from S by an application of the following sequence of algorithms:

- For some $j \in \{1, \dots, |V|\}$ the pair (i, l_i) is a result of application of LocalScheduler_j to S ;
- $\text{REMOVE}((i, l_i), Q_j)$;
- $\text{INSERT}((i, [l_i+1]_{\text{mod}(S_i+1)}), Q_{m_j(i, [l_i+1]_{\text{mod}(S_i+1)})})$.

It is known by now that the set of all discrete states of closed queuing networks is the set of all possible combinations of queue states. They can be enumerated based on the description of the queuing network by the following algorithm:

The Automaton Construction (AC) Algorithm

Input:

- A graph of the queuing system;
- For each call $i \in \{1, \dots, m\}$, the map of its steps to nodes of the graph given in the format (i, j, v) where i is a call number, j is a step number, and v is a node number of a node in queuing system graph;
- For each $i \in \{1, \dots, |V|\}$, a local scheduling algorithm. Each scheduling algorithm has the input of i^{th} -queue state and the output of a new i^{th} -queue state and a pair (i, l_i) extracted from the i^{th} -queue.

Output: The automaton with the states mapped to a set of queue states, and the alphabet being a set of all legitimate pairs of the form (i, l_i) .


```

4.   If j=1
      {4.1.   For k=1 to |Comp(j)|
            do
                s=s+1
                For i=1 to |V|
                    do
4.2.   If i=1 then
                i.key(Table(s))=GenrateNewPermutation(Comp(j),k);
                else i.key(Table(s))=i.key(current);
            od
        od
      Return}
    else
      {4.3.   For k=1 to |Comp(j)|
            do
                j.key(current)=GenrateNewPermutation(Comp(j),k);
                j=j-1
                Call GeneratePermutation(s, Table, j);
            od
      Return}

```

<generate a new queue 1, create a new state in the table by setting all queues to the same value as current adds it to the table of all states>

<generate new i^{th} queue and record it for a new state>

At the time of algorithms' termination the state table and adjacency lists for its nodes contain graph representation of a desired automaton.

Consider a Sched function for a closed queuing network and its local schedulers for all resources. Then the following statement holds:

Theorem 4.1. (Correctness) The AC algorithm constructs automata representation of Sched function from section 3. \square

Proof: Consider an arbitrary state q of a closed queuing network defined by as a set of queues $\{Q_1, \dots, Q_{|V|}\}$. The state of an automaton marked by the same set of queues must exist for the following reasons (see lines 2.1. to 2.3.):

- All m -tuples (call, step) pairs were constructed;
- For a given m -tuple states with all orders of elements in queues were constructed.

For a state q the next element i to which to apply Sched(i, q) is determined as a head element of a queue Q_i in the state that is the source of remove operation, but so is an element (call, step) in line 3.2. Therefore, for a state q the loop 3 generates all possible pairs (i, q). Sched(i, q) defines a new state q' by inserting the next step of call i into an appropriate queue; so does INSERT operation in line 3.4. Therefore, the next step must be the same. The platter proves the statement since all pairs (i, q) were considered. \aleph

For a closed queuing network with m calls each having no more than n steps the following holds:

Proposition 4.2. The AC algorithm requires $O(n^{2m}e^{2m})$ time and $O(m)$ space. \square

Proof: One iteration of the loop defined in lines 1 through 2.3 generates at most n^m m -tuples (loop line 1). Generation of m -tuple requires constant time. For each tuple at most m elements can be in a queue. To compute number of elements in queues require at most $\log m$ time (lines 2.1-2.2). For each combination of elements in queues at most $m! \sim e^m$ permutations of elements in queues are generated. Each permutation requires constant time to generate (lines 4 through 4.5). Total of $n^m e^m$ nodes were generated. Edge generating function (lines 3 through 3.5.3.2.) must consider at most $n^m e^m \times n^m e^m$ pairs of nodes. To check if the pair is an arc take constant time. Therefore, time requirements are $O(n^{2m} e^{2m})$. Accounting for space yields the corresponding space bound. \aleph

6. Timed Automata Language and Embedded Systems.

Timed Automata (see [9], [10] for details). Let $\Phi(U)$ be a set of terms in the real-boolean signature $L = \langle \{\text{Real, Boolean}\}, \{=, <, >, \leq, \geq, \wedge\}, U \rangle$, namely:

- the atomic formulas are arithmetic predicates ($=, <, >, \leq, \geq$) over variables in U , and real constants together with *true* and *false*;
- if ϕ_1, ϕ_2 , are in L , then $\phi_1 \wedge \phi_2$ also belongs to $L(U)$.

A timed automaton A is tuple $A = \langle V, S_0, \Sigma, X, I, E \rangle$, where:

- V is a finite set of vertices;
- $S_0 \subset V$ is a set of initial vertices;
- Σ is an alphabet;
- X is a finite set of variables called clocks;
- $I: V \rightarrow \Phi(X)$ is a mapping imposing time constraints on vertices;
- $E \subset V \times \Sigma \times 2^X \times \Phi(X) \times V$ is a set of arcs (called switches.) A switch $\langle v, a, \{x_1, x_2, \dots, x_n\}, \phi, u \rangle$ is an arc from vertex v to

vertex u , marked by an element of alphabet a , with the set of clocks $\{x_1, x_2, \dots, x_n\}$ reset to 0 when a switch occurs at the time when clock constraint φ is satisfied.

Semantics of a timed automaton A is defined by associating a transition system S_A with it (see [3]). A state of S_A is a pair (s, t) where s is a state of A and t is a clock interpretation for X such that $t(x)$ satisfies invariant $I(s)$ for $x \in X$. A state (s, t) is an initial state if s is an initial state and $t(x)=0$ for all clocks x . Transitions in S_A are of two possible types:

- Time transition: for a state (s, t) and $\delta \in \mathbb{R}$, $\delta > 0$ the transition $(s, t) \xrightarrow{\delta} (s, t+\delta)$ belongs to S_A if for all $0 \leq \delta' \leq \delta$, $t+\delta'$ satisfies $I(s)$;
- State transition: for a state (s, t) and an arc $\langle s, a, \varphi, \lambda, s' \rangle$ such that t satisfies φ , $(s, t) \xrightarrow{a} (s', t[\lambda:=0])$.

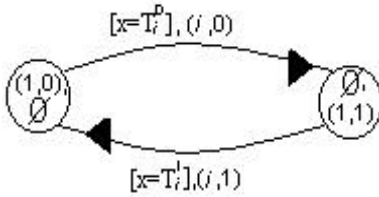


Fig. 3.

Lets start the formalization of our examples with an individual call: Each call can be defined as a timed automaton with the number of states equal to the number of steps of a call with each state labeled by by a set of states of queues; Between two consecutive steps there is a transition labeled by equation clocks defining time of transition followed by a pair $\langle i, l_i \rangle$ defining the end of corresponding event. For example in case of PLC queuing systems we have two calls of the same type (see Fig. 3.) Dynamics of the call is defined by its standard semantics. Note that standard semantics coincide with the dynamics of a call assumed in queuing system language as defined in section 5.

However, the set of calls does not uniquely determine a system and its behavior.

We need to establish a timed automaton that will define how calls cooperate producing collective behavior and at the same time will be consistent with autonomous behavior of calls. In a category of timed automata consistency with individual behavior should be interpreted as existence and uniqueness of projections from system timed automaton to each of the calls timed automata.

Since the system description is given within one category it is possible to use maps of this category do define consistency. In particular, the a time automaton QN may be considered consistent with a set of calls $\{C_1, C_2, \dots, C_n\}$ if it is equipped by projections $\pi_1: QN \Rightarrow C_1$, $\pi_2: QN \Rightarrow C_2$, ..., $\pi_n: QN \Rightarrow C_n$, and the projection of system behavior to a call coincide with autonomous behavior of a call. Even though we have not defined the projections yet, lets try to explore possible constructions. How should the time automaton look like in this case? It is obvious that a projection may take several nodes in ON into one node in some C_i . Therefore, an arc condition in C_i of the form $[x=<value>]$ when lifted along a projection to ON may be spread over a path in ON . In states along this path two alternatives could take place:

- a condition may mark one of the outgoing arcs of the state (be still possible in the state);
- no outgoing arc is marked by the condition.

The first alternative implies that a time variable for which a condition was introduced (invisible in C_i) changes value in the node. Therefore, it must be lifted to the node and to all outgoing arcs that should record the change. The second alternative means that this call is pauses in the node, its time variable does not change and it has no influence on the choice of outgoing arcs. Only presence of this variable should be recorded in the node and on arcs in this case. The second alternative is not expressible in the language of timed automata.

Suppose now that in the language of timed automata a clock is reset before entering a state q and two outgoing arcs (q, q^1) and (q, q^2) are marked by the termination conditions on this clock and nothing else. Assume are that (q^1, q^2) are not in bi-simulation relation (see[10]). The latter means informally that these states are distinguishable by runs that start in respective states. Let us try to express a timed automaton in the language of queuing theory. The only possible interpretation of time constraints in the language of queuing theory is step duration. The current state of a closed queuing network can be determined by looking at the history of (call, step) terminations, initial conditions of the system and its Sched function. It is independent of the time of termination of the (call, step) pairs as long as their sequence is known. Therefore, it must be determined by a run of the timed automaton because the run determines the history of (call, step) terminations. Initial conditions of the system must be given by initial clock settings and the starting state of the run. However, in the case that we are considering we have two runs that end in different states, but have the same history of (call, step) termination. This requires these runs to define the same state of closed queuing network. Yet this leads to a contradiction since (q^1, q^2) are not in bi-simulation relation. This informal argument can be easily formalized and leads to the following statement:

Proposition 5.1. The languages of closed queuing networks and timed automata are expressively incomparable. \square

However, our goal is to extend timed automata so that it could express anything that can be expressed in the language of closed queuing networks. We came very close with AC algorithm generating an automaton from queuing-theoretic description. The only obstacle left is to express ‘stop and go’ mode of clocks. Let us amend the definition of node constraints as follows:

$I: V \rightarrow \Phi(X) \times 2^X$ is a mapping imposing time constraints on vertices of the form $\langle \psi; \{x_1, x_2, \dots, x_n\} \rangle$ where constraint formula ψ allows to stay in the node while it is satisfied and the set of clocks $\{x_1, x_2, \dots, x_n\}$ is stopped. The rest of definition of timed automata will be the same as previously. We call the amended formalism Extended Timed Automata (ETA). Similarly to timed automata, the semantics of ETA A is defined by a transition system S_A . A state of S_A is a pair (s, t) where s is a state of A and t is a clock interpretation for X such that $t(x)$ satisfies invariant $I(s)$ for $x \in X$. A state (s, t) is an initial state if s is an initial state and $t(x)=0$ for all clocks X . Transitions in S_A are of two possible types:

- Time transition: for a state (s, t) and $\delta \in \mathbb{R}, \delta > 0$ the transition $(s, t(\mathbf{y})) \rightarrow^\delta (s, [t(\mathbf{y} \setminus \mathbf{x}) + \delta, t(\mathbf{x})])$ belongs to S_A if for all $0 \leq \delta' \leq \delta, t + \delta'$ satisfies constraint formula $\psi = \pi_L(I(s))$, \mathbf{x} is a set of stopped clocks $\mathbf{x} = \{x_1, x_2, \dots, x_n\} = \pi_R(I(s))$.
- State transition: for a state (s, t) and an arc $\langle s, a, \varphi, \lambda, s' \rangle$ such that t satisfies φ , $(s, t) \rightarrow^a (s', t[\lambda := 0])$.

We now can modify AC algorithm so its output will be an ETA defining a closed queuing network given as an input of the algorithm. As it was agreed in section 2 we limit this consideration to constant step times.

Input: The same as in original AC algorithm except the following amendment:

For each call $i \in \{1, \dots, m\}$, quadruple (i, j, v, t) where i is a call number, j is a step number, $v = m_i(i, j)$ is a node number of a node in a queuing network graph to which (i, j) is mapped, and t is duration of the step.

Modify:

```

4.2.   If i=1 then {
        i.key(Table(s))=GenrateNewPermutation(Comp(j),k);
        For k=1 to |Comp(j)|
        do
            If Next(Comp(j),k)≠First(i.key(Table(s)))    <If next element of the list is not the
            then                                           first element of ith queue then add its clock
            Stopped(Table(s))=                             to the list of stopped clocks,
            =Append(Stopped(Table(s), Next(Comp(j),k)))
            else                                           otherwise add its clock to a formula with restriction
            Formula(Table(s))=                             that it should be less than the time of the step >
            =Formula(Table(s))∪{(Next(Comp(j),i), Time(Next(Comp(j),i)))}

        od
        else i.key(Table(s))=i.key(current);

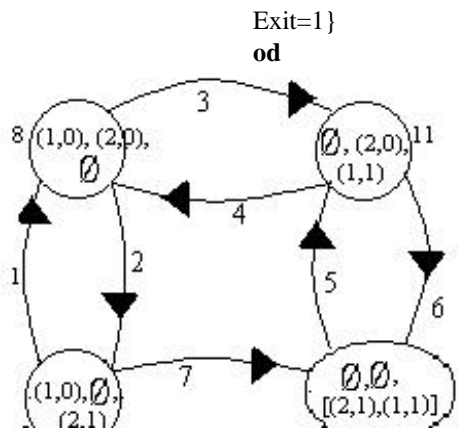
4.3.   For k=1 to |Comp(j)|!
        do
            j.key(current)=GenrateNewPermutation(Comp(j),k);
            For t=1 to |Comp(j)|
            do
                If Next(Comp(j),t)≠First(j.key(Table(s))) <If next element of the list is not the
                then                                           first element of ith queue then add its clock
                Stopped(Table(s))=                             to the list of stopped clocks,
                =Append(Stopped(Table(s), Next(Comp(j),t)))
                else                                           otherwise add its clock to a formula with restriction that
                Formula(Table(s))=                             it should be less than the time of the step >
                =Formula(Table(s))∪{(Next(Comp(j),i), Time(Next(Comp(j),i)))}

            od
            j=j-1; Call GeneratePermutation(s, Table, j);
        od
    Return}

3.5.3.2   Adjacent(StateTable(i))=
            =Append(k, Adjacent(StateTable(i));
            Formula(k,StateTable(i))={{(call, step), time(call,step)}} <transition between nodes i and k
            is marked by formula  $x_{\text{call,step}} =$ 
            =timecall,step>

            Newclock(k,StateTable(i))={{(call,next)}}
            <transition between nodes i and k is
            Mark(k,StateTable(i))=(call, step)
            Is marked by new clock  $x_{\text{call,next}}$  and element
            of alphabet (call, step).

```



Exit=1 }
od

If Exit=1 exit the loop
od

od

Modifications to 4.2 and 4.3 guarantee that the restriction formula and the list of stopped clocks will be formed correctly for each state. Modification to 3.5.3.2. do the same for transitions between nodes. Apparently, these

modifications do not increase the complexity of the algorithm. We now apply Modified AC (MAC) algorithm to our original example. The result is on Fig. 4. Here the nodes and arcs are marked as follows:

1~ $[x_{1,1}=T_1^1], \{x_{1,0}\}, (1,1)$; 2~ $[x_{1,0}=T_0^1], \{x_{1,1}\}, (1,0)$; 3~ $[x_{2,0}=T_0^2], \{x_{2,1}\}, (2,0)$; 4~ $[x_{2,1}=T_1^2], \{x_{2,0}\}, (2,1)$; 5~ $[x_{1,1}=T_1^1], \{x_{1,0}\}, (1,1)$;
 6~ $[x_{1,0}=T_0^1], \{x_{1,1}\}, (1,0)$; 7~ $[x_{2,0}=T_0^2], \{x_{2,1}\}, (2,0)$; 8~ $[x_{1,0}<T_0^1] \& [x_{2,0}<T_0^2], \emptyset$; 9~ $[x_{1,1}<T_1^1] \& [x_{2,0}<T_0^2], \emptyset$;
 10~ $[x_{1,1}<T_1^1] \& [x_{2,1}<T_1^2], \{x_{2,1}\}$; 11~ $[x_{1,0}<T_0^1] \& [x_{2,1}<T_1^2], \emptyset$.

Let N be a closed queuing network with constant step times. Let also ETA MAC(N) be its transformation by MAC algorithm. Consider a trace of a run of associated transition system $S_{MAC(N)}$ with a given set of initial clocks and a maximal interpretation time and the simulation output of CONS(N) with the same initial conditions and the same maximal time of simulation. Then the following statement relates a trace of a run and simulation output:

Theorem 5.2 Under the above conditions the simulation output of CONS(N) coincides with the trace of the run of $S_{MAC(N)}$. \square

Proof idea: By induction on run length. Technical details are omitted because of space limitations. $\frac{3}{4}$

Corollary 5.3. Simulation of a closed queuing network behavior for up to n events can be done in time $O(n)$.

Proof: immediate from theorem 5.2.

6. Discussion

In this paper we introduced a new formalism for modeling embedded systems. We have shown that this formalism is at least as expressively powerful as the formalism of closed queuing networks. Moreover, we have shown that using this method we can simulate network behavior faster than using traditional simulation approach (in real-time). In fact most of the work lies in preprocessing: reformulation of a network description from the language of queuing theory into ETA formalism. It is interesting to see if methods of analysis developed for timed automata may help in analysis of closed queuing networks.

References

- [1] ISO/IEC JTC1/SC7/WG11. High Level Petri Net Standard. Committee Draft, 1998.
- [2] A.B. Wolpert. On Implementation of Time in Dynamical Modeling of Computer Systems: High Level Petri Net Approach. In Proceedings of IASTED/ISMM International Conference on Modeling and Simulation, MS 99. 1999, pp. 224-230.
- [3] R. Alur. Timed automata. In Proceedings of NATO-ASI Summer School on Verification of Digital and Hybrid Systems, 1998, Pp. .
- [4] R. van Glabbeek, P. Rittgen. Scheduling Algebra. Proceedings of the Seventh International Conference on Algebraic Methodology and Software Technology (AMAST '98), Amazonia, Brazil, January 1999 (A.M. Haeberer, ed.), LNCS 1548, Springer, 1999, pp. 278-292
- [5] R. Gorienni, M. Rosetti. E. Stancampiano. A Theory of processes with Durational Actions. Theoretical Computer Science, v. 140, n. 1, 1995, pp 73-94.
- [6] A.A. Borovkov. Asymptotic methods in Queuing Theory. Wiley, NJ, 1984.
- [7] J. Banks, J.S. Carson, II, Barry L. Nelson. Discrete-Event System Simulation, Prentice Hall, New jersey, 1996.
- [8] E. M. Calrke, O. Grumberg, D.A.Peled. Model Checking. MIT press, MA, 1999
- [9] R. Alur, T.A. Henzinger., A really temporal logic., Journal of the ACM, V. 41, No. 1
- [10] G.J. Holzmann. Design and Validation of Computer Protocols. Prentice Hall, NJ, 1990