A TIMED COLORED PETRI NETS APPROACH TO PROCESS SCHEDULING
A. Camurri, P. Franchi and F. Gandolfo
(IEEE Student Members)
DIST - Department of Communications, Computer and Systems Sciences
University of Genoa, Via Opera Pia 11A, I-16145 Genoa, Italy
E-mail: music@dist.unige.it

Abstract
This paper introduces an algorithm for solving multiprocess scheduling problems, based on an extension of Petri Nets (PNs) able to cope both with multiple processes and with time constraints. These nets are called Timed Colored PN (TCPN). The class of problems faced by our approach can be characterized as follows: there is a set of concurrent processes, each formed by a number of temporally related tasks (segments). Tasks are executable by alternate resource sets. Processes and tasks are characterized by release times, due dates and deadlines. Time constraints are also present in the availability of each resource in resource sets. It has been proved that the problem does not admit an algorithm for an optimal solution in a polynomial time. The proposed algorithm finds a suboptimal schedule according to a set of optimization criteria, based on tasks' and processes' times (earliness, tardiness), and/or on time independent resources' costs.

1 Introduction
This paper introduces an algorithm for solving multiprocess scheduling problems, based on an extension of Petri Nets (PNs) able to cope both with multiple processes and with time constraints. These nets are called Timed Colored PN (TCPN). The class of problems faced by our approach can be characterized by a set of concurrent processes, each formed by a number of temporally related tasks (segments). Tasks are executable by alternate resource sets. Time constraints are present both in the availability of each resource in resource sets and in processes and tasks. It has been proved that the problem does not admit an algorithm for an optimal solution in a polynomial time. The proposed algorithm finds a suboptimal schedule according to a set of optimization criteria, based on tasks' and processes' times (earliness, tardiness), and/or on time independent resources' costs.

Process scheduling is a most general problem; situation in which such a problem arise can be found in many research fields as well as in practical applications. Communication networks dealing with packet switching should find a possibly optimum routing policy in order to minimize queue size and time delays. Hard-real-time systems [8], too, show the need of efficient scheduling strategies since they deal with precedence relations between tasks, for example if a task requires information produced by another one. Time constraints are also important to synchronize the access to shared resources such as data or I/O devices. Such problems may also be found in the management of flexible manufacturing systems (FMS) [6] and in decision support systems [4], where it is known as the project management problem. Process scheduling finds useful application also in non classical domains. For example, since computer music performance systems [11] [7] are defined as “computer systems connected to input devices (including musical keyboards or other real time instruments) and to graphics and audio output devices” [8] they require real time scheduling strategies for the management of events from input devices and of actions to be performed.

The leading example analyzed in the paper is related to the coordination and the scheduling subsystems for the simulation and the real-time control of FMS. Several hypothesis have to be kept into account in the design of efficient schedulers. Xu and Parnas [17] recently developed a preemptive scheduler for multitask applications on a single processor. Our approach starts from a different hypothesis: it allows the scheduling among a number of processors but it is not preemptive, as in most industrial machines. To do this, we propose the architecture shown in figure 1. This is basically a two-level architecture. The approach followed in this paper, in fact, clearly distinguishes the mechanism of net execution from the decision support system. Basing on this fact, two conceptually distinct levels (corresponding to two different, interacting implementation modules) have been developed: the executor, and the scheduler level. Referring to an FMS context, taken as a case study in the paper, these levels correspond to the coordination and the scheduling subsystems.

One of the outstanding differences between these levels is that the executor is conceived as a sort of "slave" component: it is a fast, efficient Colored PN executor, without any problem-solving capability in case of conflicts. The scheduler, on the other hand, is the active, adaptive counterpart, and its structure and behaviour may heavily depend on the problem instance, even if its general architecture can be adopted in a number of possible cases.

2 Problem definition
This section defines the process scheduling problem, referring to our FMS example. Basic terminology and some definitions are borrowed from [4]. The problem has its primary focus in the scheduling of concurrent processes, where a process is a set of temporally related tasks, whose performance gives a complete production cycle of a workpiece in a FMS. Different processes correspond to different production cycles for different workpieces, and processes may be subject to conflicts in the use of machines and resources. In general, different pallets will be necessary for each workpiece type in the FMS. Therefore, the number of different pallets available in the FMS defines the maximum number of processes which can be carried out. A set of processes has to be run simultaneously, that is different types of workpieces must simultaneously be processed by the FMS. Tasks' activities are defined in terms of functions. A task can be assigned to one or more machines (stations) in the production line. Machines are a class of possible resources; in general, a task can be accomplished by different resource sets (RSs). The decision on the resource set for
a given task instance at a given time is demanded to the real-time scheduling subsystem, which operates according to the optimization criteria and the constraints defined in the problem. Each resource is characterized by an Availability Table, which specifies the time intervals in which the resource is available. The architecture of the FMS should allow the continuity in the production process, even in case of unavailability (programmed or not) of machines. Couples of tasks of a given process are linked by temporal relations. In general, they specify either similar precedence relations, with possible definitions of waiting times between the ending instant of a task and the beginning of the next, or more complicated explicit time relations: for example, the overlapping of the completion of a task with the beginning of the next. In a FMS domain, the latter case is generally not necessary, since

Figure 2: A production line. a workpiece is generally processed only by one task at a given time. In any case, our approach allows general temporal relations among tasks. Waiting times between two sequential tasks may have soft constraints: for example, a workpiece after a painting task may wait for a time interval from 10 minutes to 30 minutes, before its processing by the next task. This is another typical decision situation: the coordination subsystem demands all decision situations to the real-time scheduling subsystem, which should determine the optimum value for a given case, basing on specified criteria. In figure 2 a simple example of FMS scheme is shown. The example regards two processes to be carried out. Each process consists of a sequence of two tasks for a different workpiece. For both processes, the first task can be accomplished using either machine M1 or machine M2; in both cases a further resource R1 is needed. Therefore, Task 1 has two possible task instances (T1) and (T12), corresponding to the use of M1 or M2, respectively. The same situation occurs for Task 2, with the only difference that in this case we have three TIs (T121, T122, T123), and machines M3 and M4 are needed from T122, so they appear in two different RISs. We hypothesize that each resource in the example (generic or machine type) has an availability table, consisting of a set of couples of time points, defining the time intervals in which such resource is available. Yet, a waiting time is needed between the end of task 1 and the start of task 2, for both processes. Each process has different waiting time ranges.

A production line in which different processes are concurrently processed needs a scheduling policy to optimize the productivity, basing on criteria and costs of some kind. We hypothesize that tasks and processes have constraints on their workpieces' release time, due date and deadline, respectively the initial time at which the task or process could start, the due date for its completion, and the maximum time allowed for its completion. Optimization criteria, based on tasks' and processes' times, or on time independent machines' costs, are taken into account in the problem's formulation (see further in the paper).

3 A Timed Colored Petri Net-based Coordination Subsystem

Petri nets find useful application in the modeling of systems characterized by a distributed and concurrent nature, by the synchronization among tasks in the use of shared resources. [13, 14, 15]. In particular, PNs allow a structured definition of the model, reflecting the nature of the problem. This leads, for example, to faster implementation and maintenance of the simulation and control subsystem’s model, of the scheduling strategies and of the consistency checking procedures.

Furthermore, in the last decade, new formal extensions to the original PN model (P-T nets) played (and are still playing) a fundamental role in the diffusion of PNs in several fields, such as FMS and Operations Research. A particular usefulness of PNs has been recently pointed out in the field of FMS [12]. Timed PNs and Stochastic PNs [1, 2], Colored PNs (CPNs) [18] and Predicate/Transition nets (Pr-T nets) [9] are among the most significant extensions, and are currently used in many application fields. CPNs and Pr-T nets are generally known as High-Level PNs (HLPNs), since the proved equivalence of their expressive power. A P-T net can be described by a considerably reduced (in size) HLPN; that is, HLPNs are a powerful tool for the definition of models of concurrent distributed systems at higher level of abstraction. This fact is of considerable help in the modeling of FMS, typically involving a large number of resources and tasks. HLPNs allow the definition of individual tokens, and transitions behave in different ways according to the firing color (selected from the input places): a transition in a HLPN corresponds to a set (a class) of transitions in the P-T nets model. This can be considered a particular "object oriented" approach, in which classes of tasks, corresponding to a transition, are connected to their related resource sets (set of places). A different approach for introducing object-oriented concepts into PN theory can be found in [9]: the authors define objects as subnets (with a fixed topology) in an extended P-T net model (PROT nets); high level transitions are objects representing subsets.

In our model, we adopt an extension of Colored PNs, embedding time transitions (TCPN); their basic features are now briefly discussed. Colors correspond in our model to the different processes: at a given time, a process state is identified in a TCPN by the set of tokens of its color. The flow of tokens of a given color in the TCPNs model represents the evolution of the associated process. A neutral color is defined: a resource (place) marked with a neutral color can be utilized by any process, that is, a neutral colored token can be used in any colored firing occurrence. Transitions correspond either to TIs (TI-transitions) or to waiting processes (W-transitions), and are characterized by a processing time, which can be deterministic or stochastic. Immediate transitions (with a null processing time) are allowed; their default management in case of conflicts follows [1], that is, conflicts between an immediate and timed transition are always solved in favor of immediate transitions.

The information on the processing time of a TI-transition is particularly useful in case of system’s simulation. In a real world environment we have real (low level) local controllers, each of them linked with a suitable TI-transition at TCPN level: the firing of a TI-transition causes the sending of messages to its controller for the activation of the corresponding real task instance; the expiring of the transition can be driven by the completion of the task instance, via messages coming to the transition from its controller. In a simulated environment, we know the parameters of the machine(s) executing the task (for example the speed) associated to a TI-transition, and we can use them for computing its processing time, either deterministic or stochastic. W-Transitions may have soft-constrained processing times: only the range is known. The real-time scheduling subsystem will decide a suitable value in the defined range for each soft-constrained W-transition. Other basic properties of transitions are a status, the set of the allowed firing colors, and a set of functions defining the code for the local controllers: for each transition, there is a function for each color (i.e. process) in the net.
Places correspond to either resources or queues. A place is characterized by a possible availability table and by its set of tokens. The availability table typically models sharing features of places.

The firing rule is the same adopted in CPNs, with the exception of the delays due to the transition’s processing time. In fact, a transition firing causes the consuming of input tokens, and only after the transition’s processing time has expired the output tokens are placed on the output places. This causes tokens to appear/disappear in the net: they are stored in transitions for the duration of their processing time. This is a bad feature, from the point of view of analysis properties: it is possible to reconduce this behaviour to the standard CPN model if we consider a (timed) transition as a particular subnet, simply formed by a sequence of transition-place-transition (see figure 3). These transitions are immediate: the first one consumes input tokens, and marks the intermediate place; after the processing time, the final transition fires (it is the firing rule which changes, now), consumes the token from the intermediate place and produces output tokens. Therefore, any timed transition in our PN model can be seen as a sequence transition-place-transition.

As for the firing rule, a transition fires if all input places are marked with a given color, or some places are marked with the neutral color and the remaining places with the same color. The firing color must belong to the set of transition’s allowed colors. Multiple firing of the same transition, due to the availability of multiple pre-sets, is generally not used in this model, even if it is allowed by the HLPN model. A further condition has been added to the firing rule of transitions, because of the presence of availability tables on resources: an input place is available if it is marked and at the current time, and within a ‘reasonable’ time horizon, the resource is available from its Availability Table. The ‘reasonable’ horizon is deterministic in the case of FMS simulation: it is the time point obtained adding the current time to the processing time (computed for the firing color) of that transition.

Figure 4 shows the subnet related to the Task 1 of the previous example of figure 2.

Note that this net could be collapsed into a smaller TCPN, using the properties of HLPNs: all TIs could be represented by means of a single transition with a slightly different firing rule, allowing multiple concurrent firing for disjoint pre-sets (see [10]). We have chosen in this case a representation as in figure 4.

Figure 5: The Petri net for the production line of fig. 2, with separate transitions, because of its major resemblance to the problem’s nature, its easier understanding and better tractability. In other cases we use multiple firing transitions. The complete TCPN for the example of figure 2 is shown in fig. 5

4 The executor subsystem

The executor subsystem (modeled as a TCPN) is the module dealing with the net topology; its function is to present the scheduler with the transitions enabled to fire and to drive the tokens’ flow along the net.

The first hypothesis we made, to cope with real world situations, is that each resource is not available for the whole simulation time, but there exist some time interval in which it cannot be used, for example due to periodic machine maintenance; it is then possible to priori build a table indicating, for each resource in the problem, the time intervals in which it is available (Availability Table).

As a second hypothesis, we suppose that time is divided into equally spaced atomic slices, and in some relevant slice (inserted into a structure called Time Table), transitions are tested for possible execution. The instants we consider relevant are:

(i) the initial time;

(ii) the time in which a resource becomes available, according to its Availability Table;

(iii) any time slice in which a transition completes its activity, i.e. able to mark its output places.

The first two cases are dealt with during the initialization of the tables, while the third one causes a dynamic update of the table according to the net behaviour. At each significant time slice, each transition is tested to check if it is enabled to fire and, being the test successful, the code associated to the firing rule is executed.

Each execution slot can be divided into three parts: first, immediate transitions (with zero time duration) are executed (xtrans0 procedure), then all the other selected transitions fire but do not (yet) produce output tokens (xtrans1 procedure); time goes on one slot and, finally, expiring transitions mark their output places (xtrans2 procedure).

The following listings show the details of the executor’s structure (trans stands for a generic transition.)

While there are events in the time table, xslot_start is executed and the relative output places are marked.
xslot_start() /* starting time slot execution module */
{
    // (firable immediate transitions) xtrans0(trans)
    for all trans xtrans1(trans) consume input tokens
}

"for all trans" is a short expression for the algorithm dealing with
the order of examination of the transitions: in each time slot, all
transitions are scanned in a random order by a hashing routine; this
avoids unwanted priorities among concurrent transitions, al-
lowing a fair simulation policy.

It is worth noting that the difference between immediate and
"normal" transitions is not limited to the duration alone, but
it involves a different behaviour as well. While the latter may
fire at most once in a time slot, no limits are imposed as to the
activity of zero duration transitions. This allows some classes of
tokens to shortcut some regions of the net not needed for they
purposes. This way no additional delays, due to the topological
structure of the net rather than to the physical meaning of the
problem, is imposed to these tokens. As it can be seen from the
implementation code in procedure xslot_start, loops involving
immediate transitions alone are forbidden, since they result in an
endless loop; this hypothesis, on the other hand, seems not to
limit the potential of our approach, since we have not been able
to find a reasonable situation in which such a configuration is
needed.

xtrans0(trans) /* immediate transitions management */
Transition trans;
{
    if (the duration of trans is not zero) return
    Color = select(trans)
    compute the firing probability of trans
    if (probability is 0) return
    for all input places of trans {unmark the place}
    for all output places of trans {mark the place}
}

xtrans1(trans) /* first phase of (normal) transition management */
Transition trans
{
    if (the duration of trans is zero) return
    Color = select(trans)
    compute the firing probability (P) of trans
    if (P is zero) return
    else if (P == -1) {/* there is a conflict */
        trans = T_real_time_scheduler(trans)
    }
    execute transition function code
    send activation message to trans' controller
    for all input places of trans {unmark the place}
}

select(trans)
Transition trans
{
    check firing rule and availability tables
    choose the firing color for trans checking
    if it exceeds the release time
    if (no color available) return failure
    if (trans has more than one firing color)
        Color = C_real_time_scheduler(trans)
    return Color
}

xtrans2(trans)
Transition trans
{
    if (the duration of trans is zero) return
    for all output places of trans {mark the place}
    if (beyond deadline) simulation is aborted
    exit with an error message
}

The probability of a transition to fire is a function depending
on the status of its input places. On the other hand, the fact that
each input place may affect different transitions must be taken
into account. This results in a more complex computation of the
firing probability of each transition. The following procedure
shows our way to compute the probability function.

probability(trans)
Transition trans
{
    double prob = 1.0
    for each input place of trans
        prob = prob * prob_place(p, trans)
    if (real_time_scheduler available)
        if (0 < prob < 1) return -1
        /* a topological conflict has been detected */
        /* deterministic or random scheduling: */
        random = pick up a number between 0 and 1
        if (random < prob) return 1
        else return 0
    }
    The probability of a transition to fire in respect to the place p is
a number between 0 and 1 computed as follows:

prob_place(p, trans)
Place p Transition trans
{
    int counter = 0
    for each transition in output of p except trans {
        if (transition already scheduled) continue
        if (transition can fire)
            counter = counter + 1
    }
    if (counter is not zero) return 1/counter
    else return 1.0
}

5 The scheduling subsystem

Petri net theory does not embed any conflict-solving method: the
firing rule does not specify when a transition fires; it only specifies
that a transition may fire if the firing rule is satisfied. Then, the
decision on which firing rule to adopt plays a fundamental role in
the net evolution. Conceptually, the firing rule can be figured as
a decision level above the executor, gaining though the status of a
scheduler. Its function is twofold:

(i) conflict solving;

(ii) guiding the net towards the (sub)optimum timing.

These two aspects are, of course, deeply interconnected since they
both rely on the choice of a path at a decision point, which in most
cases, is a conflicting situation. The following subsections offer an
overview of the possible conflicting situations and of the related
schedulers.

5.1 Conflicting situations

The inner structure of the considered PN allows for some critic
situations to arise. They can be classified in the following cate-

307
(i) a topologic conflict (figure 6.a),
(ii) a color/project conflict (figure 6.b),
(iii) a duration indeterminacy (typically W-transitions),
(iv) mixed situations, typically simultaneous (i) and (ii).

A topologic conflict arises when a single token could enable more than one transition in a mutually exclusive way: a decision should be taken, as to which transition to select. Back to our FMS trailing example, this means that there exist at least two machines able to process an incoming pallet; therefore, a policy selecting the most suitable processing cycle for that particular pallet is needed. A colour conflict, on the other hand, arises from the opposite situation: a single transition may be enabled, at a given time, by differently coloured sets of tokens. In this case the decision depends not only on each token in itself but also on its semantic, since the decision should be affected by the role the token plays in the specific instance of the examined problem. This means that a single machine could be faced with the problem of finding several pallets ready for processing and having to choose which of them should be given precedence.

The third situation (wait conflict) involves transitions with soft constraints, i.e. transitions whose durations is not a priori set but can be varying within a fixed range. In this case, since the net is intrinsically deterministic, a decision should be taken as to the actual duration of each specific transition. Soft constraints are used in FMS language to give the decision system an additional degree of freedom in order to minimize input queues for the machines. Queues may in fact cause stocking problems in real applications, and they should be avoided to limit the cost of additional stockage. Of course, some of the above described situations may occur simultaneously, and a cooperation mechanism among the strategies should be taken into account.

Since the optimum solution is generally not available in a reasonable amount of time (due to the combinatorial explosion of the problem), suboptimum local schedulers must be devised for real world applications.

5.2 A Montecarlo-like general-purpose scheduler

The easiest, general purpose scheduler which could be conceived is a Montecarlo-like schema (figure 7). The PN is run allowing random selection of the enabled transitions as a firing rule.

The output of the simulation is then evaluated by a special module which decides, on the bases of previous evaluation, whether to rerun the net or to output the current best result. This decision could be taken either by setting a fixed number of iterations or by constraining the variance of the simulation within a limit, or by any other method. The basic advantage of this schema is its flexibility, since it is problem independant. This, on the other hand, results in poor performance because of the trade off simulation time versus quality of the solution. The more iterations are allowed, the better solution is found (apart from some degenerated situations). Its most outstanding disadvantage is that it is not feasible in a real time environment, where time limits are tight and a good decision is to be found in a short fixed amount of time. Then, this method can be used for off-line planning alone, leading to a most probably suboptimum schedule in a real world application, since it is hard to have precise a priori information. While, on the other hand, overall statistics can be provided.

5.3 Special purpose schedulers

In this section we describe some problems arising in the definition of special purpose schedulers, taking the FMS case study as an example. Color conflicts are examined first. Since it may happen that several pallets are ready to be processed on the same machine at the same time, a decision on which pallet to let first should be taken. Our choice has been to give precedence to the pallet with the most tight schedule in order to slow down the production cycle by letting a pallet wait and run out of time.

From the implementation point of view, the colour scheduler is realized by means of the function $C_{real\_time\_scheduler}$, which takes into account the pool of conflicting pallets and returns the selected one. There are three criteria adopted in turn to reach the best choice: at first, the pallet with the closest due-date is chosen; on the other hand, if no due-date is specified for any pallet, the deadline is taken into account. As a last resource, random choice is allowed, that is for if no expiring dates are specified, all the pallet can be considered to hold equal priority.

In other words this is expressed as:

- $first\_due = project\ whose\ due\ date\ is\ set\ and\ closest$
- $first\_dead = project\ whose\ deadline\ is\ set\ and\ closest$

If deadline is specified for at least one project

- $return\ first\_due$
- $return\ first\_dead$
- $return\ a\ randomly\ chosen\ project\ in\ the\ pool$

Another conflicting situation (a topologic conflict) could arise if two or more machines were competing in order to get a pallet ready for processing; this means that only one of the involved transitions in the net must be allowed to fire. This situation presents of course no ambiguity as to which job to choose, since any such conflict has already been solved by the colour scheduler: instead, a decision is to be taken as to which transition to activate. In FMS language, it can be stated that a task could be performed in several different ways and the most profitable combination of human and machine resource must be found. Our approach is to fire the fastest transition, provided this decision does not damage any incoming job. In other words, if one of the transitions immediately above the conflicting one is processing a job which should be given precedence due to its time limits, then it may be chosen to leave the fastest transition to that job and to fire the next fastest transition for the current job. This is not of course the only choice that could be made; for example, heuristics could be designed in order to take into account the resources' costs and to combine it with the deadline of the project.

Its implementation is straightforward: the procedure $T_{real\_time\_scheduler}$ is invoked by each transition. If there exist conflicting transitions, they are detected and among them, the two transitions with minimum duration are selected. The upper
transitions are then examined to see if there exist an incoming job which is likely to be running out of time and which could take advantage of finding the quickest transition free waiting for it. If such a job is found, then the current job is assigned the next fastest transition, otherwise, the fastest transition is chosen.

The discarded transitions are then marked as already examined, to save computation time; in this way, they will not be taken into account anymore as likely to be scheduled for the current time slot.

In pseudocode this is expressed as follows:

```
detect the conflicting transitions
select the fastest two transitions
check for an incoming job which should be given precedence
mark discarded transitions as already considered
return firing transition
```

The most general problem which could be solved by the net includes also the definition of compulsory time delays between consecutive tasks. Such delays can be generally defined as a time interval indicating the minimum and maximum delay allowed between the tasks. A decision should be taken, in a deterministic environment, as to which duration to adopt, whether the minimum or the maximum or maybe an intermediate one. Since Petri nets impose time delays on the jobs on their own account (there can be conflicts or the desired transition could be already busy), it is reasonable to adopt as fixed time delay the minimum. This choice seems to be satisfactory also from the theoretical point of view, since it minimizes the probability that the sum of net's delay with the fixed delay would exceed the maximum interval allowed.

From the implementation point of view, the procedure $W_{realTimeScheduler}$ only assigns the minimum time delay to the transition, instead of a random duration.

**Conclusion**

Our approach to problem modelling via TCPN and process scheduling has been presented.

Extensions to classical Petri nets have been considered. A terminology for the specification of process scheduling problems has been presented.

Taking FMS as a case study, we have investigated the possibility to find a suboptimal scheduling policy according to preselected criteria. We have devised a two-level architecture in which the Petri net executor acts as a slave and the upper level takes care of decision making. We have considered a general purpose Monterarto-like scheduler and a special purpose problem driven scheduler for our case study.

The result of our research is a discrete event simulator called Petri+ able to take a process scheduling problem as an input and to produce a feasible schedule as an output. Such schedule may be interactively modified by the user and the simulator is able to verify the feasibility of the modified schedule.

**Acknowledgement**

This work is partially supported by Special Projects on Robotics and Artificial Intelligence of the Italian National Council of Research (CNR). We are grateful to Renato Zaccaria and Marcello Frixione, for the stimulating discussions.

**References**


