Control and Cybernetics

vol. **35** (2006) No. 3

A timed Petri net framework to find optimal IRIS schedules

by

Matthias Werner

TU Berlin, Communication and Operating Systems Group
Einsteinufer 17, 10587 Berlin, Germany.
e-mail: mwerner@cs.tu-berlin.de

Abstract: IRIS (increasing reward with increasing service) real-time scheduling appears frequently in real-time control applications such as heuristic control. IRIS requires not only meeting deadlines, but also finding the schedule with the best result (highest reward). In this paper, a framework is presented that uses Timed Petri nets (TPN) to transform an IRIS problem into a dynamic programming (DP) problem, allowing the application of known TPN and DP techniques. In the presented approach, an IRIS problem with tasks having discrete-time optimal parts is transformed into a (possibly unbounded) TPN. Then, the critical path problem of the TPN state graph can be tackled with DP. This approach allows for the IRIS problem multiple constraints and negative rewards.

Keywords: IRIS, increasing reward with increasing service, scheduling, Timed Petri nets, critical path, real time.

1. Introduction

1.1. IRIS scheduling

In real-time computing, the value of a computation does not only depend on a correct outcome, but on the time the outcome is computed. Thus, when computation operations in that domain are scheduled, timing constraints have to be considered. These timing constraints are usually given in terms of deadlines.

There exists a number of computation problems, where a longer run of the computation provides better result. Consider for example the computation of the value of π: there are several iterative approaches that provide a more precise result when running more iterations, i.e. with increasing run time. Other examples, where longer run time provides better results, are search problems (e.g., chess game) and several heuristic algorithms.

The problem domain described here is called *imprecise computing* or *increased reward with increased service*, short IRIS. In this paper, we introduce a

class of IRIS problems that is based on a set of assumptions, which are frequently met in real computing systems. We present a framework to model problems of this class with Timed Petri net, and we discuss how to apply known methods of dynamic programming.

1.2. Related work

Publications about Petri nets (Petri, 1962) are legion, even if one restricts the area of interest to time-dependent Petri nets or even Timed Petri nets (Ramchandani, 1974). A good overview can be gained by using the Petri net Bibliography maintained by the University of Hamburg (Rölke, Heitmann, no date). Applications of Petri nets in scheduling are numerous. Targeted problems are, among others, deadlock avoidance, performance evaluation, or feasibility of real-time and non-real-time problem. We are not aware of any Petri net application in IRIS scheduling.

In comparison with Petri nets or general scheduling, publications on IRIS scheduling (Dey et al., 1998; Krishna, Shin, 1997) or imprecise computing (Liu et al., 1991, 1994; Shih and Liu, 1995) are rather infrequent. Krishna and Shin (1997) gives a good overview. Yee and Ventura (1999) use dynamic programming to solve an assembly line scheduling problem described by Petri nets. In Popova-Zeugmann and Werner (2005) the shortest/longest path problems are used to find optimal real-time schedules, where the task systems are modeled by Time Petri nets. The problems targeted in both papers differ from the problem here: there are no optional task parts as in our approach (see Section 2.1).

1.3. Paper structure

The rest of this paper is organized as follows: Section 2 discusses the IRIS problem and introduces assumption for an IRIS scheduling in a realistic environment. Section 3 defines Timed Petri nets. In Section 4, the modeling elements of the framework will be introduced. The spawning of a modified state space and the application of critical path (longest path) algorithm to find the optimal solution of the IRIS problem is discussed in Section 5. In Section 6 a demonstation of the approach is presented, using an simple example system and a modified Bellman-Ford algorithm as critical path algorithm. The paper concludes with a short summary in Section 7.

2. The IRIS problem

2.1. General problem

IRIS task scheduling is a problem, which appears in certain real-time systems.

Real time system in a general sense are systems that have to deal with (external) timing constraints. Usually, the units of executions in a computer systems are tasks. Then, the timing constraints exist for the task set, and the

constraints are expressed in form of deadlines. It is possible that every task have its own deadline, but frequently, there is the same deadline for any task in the task set. After the deadline, no task execution is allowed any more.

Beside the time constraints, there are other constraints that may limit the execution of tasks: a task needs resources during its execution. The most important resource is a CPU that executes the tasks. If there is one CPU only, only one task can be executed at any instance of time. In turn, during its actual execution, a task occupies the whole CPU. (In a time sharing system, not all time between the start and the end of a task belongs to task execution time: the task execution may be suspended, e.g., to allow another task to run). Other resources, e.g., sections of memory or input/output devices, are merely needed for a short time during task execution. It is also possible that the computation of a task requires the results of other tasks. Then, one says that this task depends on the other tasks, and its execution can not be started before the other task executions end.

Beside the external constraints, there are conditions that characterize the task execution itself. First, it may be possible that a task execution may be interrupted or not. If the first is true, task execution will continue until it is finished, once started. Most Real-time systems allow interrupting and resuming the execution of tasks.

If a system has more than one CPU and interruption is possible, a task may migrate during interruption and resume its execution on another CPU. However, there are only very few systems that support task migration, and almost none of them is real-time.

Until this point, we have talked about real-time systems in general. Now we want to discuss how IRIS systems differ.

Beside a deadline, for each task in an IRIS system there exists a reward function. This reward function gives for a task T_i the reward that is gained when the T_i is executed for a time Δt. Typically, the reward function is of a form

$$f_i(t) = \begin{cases} -\infty & \text{if } t < t_m \\ r(t) & \text{if } t_m \leq t < t_m + t_o \\ r(t_m + t_o) & \text{if } t_m + t_o \leq t \end{cases} . \tag{1}$$

The part of the task that is executed after the task start until t_m time units have passed is called the *mandatory* part of the task. (This refers to task time, not to wall clock time, i.e., if the task execution can be interrupted, the task time ceases to progress during an interruption). The part of the task after the mandatory part, i.e., within the interval $[t_m, t_m + t_o]$, is called the *optional* part.

Frequently, the reward of the mandatory part of (1) is given as 0. However, since real-time systems often fail, when any mandatory part of a task miss the task deadline, it makes sense to give a penalty, so that optional parts of other tasks can not compensate. Other cases that occur are that t_m is zero or t_o is

infinite. The first case means that the task has no mandatory part, i.e., skipping the task is not critical. In the second case, each increment of the task runtime has an impact on the reward. To illustrate this second case, please consider the example from the introduction of calculating the value of π. If an algorithm like the one described in Press et al. (1992) is used, each round of calculation provides a more precise result (with an assumed higher reward).

If there are more than one task in an IRIS system, the total reward is assumed to be the sum of the single rewards. It is possible to use a weighted sum, but we assume that the weights are already considered within the task reward functions.

Now, we are ready to form the IRIS scheduling problem:

DEFINITION 2.1 (IRIS scheduling problem)
For a set of IRIS tasks $\{T_1, T_2, \ldots, T_n\}$ *and a set of constraints* $\{C_1, C_2, \ldots, C_m\}$ *find a schedule that meets all constraints* C_j *and provides a maximal reward.*

Obviously, the IRIS problem is an optimization problem. It is easy to see that the general IRIS problem is $\mathbb{NP}$-hard. However, there exist a range of special instances of the problem that are solvable in polynomial time; for some of them exist even on-line algorithms. Examples are IRIS tasks sets with linear reward functions or task sets with identical concave reward functions, both at single processor systems with no task interdependencies.

2.2. Real environments

If we take a closer look at the "real" real-time systems, we will find a number of properties that are often neglected by IRIS approaches.

- R.1 **discrete scheduling quantums.** In real environments, tasks will neither be interrupted at arbitrary times, nor they will run for arbitrary short time intervals. Rather, there is a minimal time quantum of execution depending on the actual system.
- R.2 **discrete reward steps.** In real environments, the optional parts of a task consist of discrete subparts. Only if such a subpart is successfully executed, one gains a reward. This is at least due to the minimal scheduling quantum. Moreover, an IRIS task usually works round-based, where only a finished round contributes to the reward.
- R.3 **task interdependencies.** In real environments, tasks frequently share resources or depend on results of other tasks.
- R.4 **partly decreasing reward functions.** It is not very common, but it is possible that a reward function is not (as always assumed) monotonically nondecreasing over the time. Sometimes, starting a new round may consume reward (e.g., if external resources are consumed); thus only a successful computation round provides a positive reward.

The approach described in this paper assumes R.1 and R.2 and allows R.3 and R.4. In addition, we assume a common deadline for the whole task system. This

condition is frequently met in real systems, since the different tasks of a task set contribute to the same problem. However, the presented approach can be extended by additional instrumentation that lift this condition.

Other conditions of real environments, such as context switch times or communication overhead are neglected in our framework. However, extensions to consider these conditions seem to be easily implementable.

Considering R.1 and R.2, a task reward function has the following form:

$$f_i(t) = \begin{cases} -\infty & \text{iff } t < t_{i,1} \\ \sum_{j=1}^{n_i} r_{i,j} H(t - t_{i,j}) & \text{else} \end{cases} \tag{2}$$

with $\forall j = 1 \ldots n_i - 1, t_{i,j} < t_{i,j+1}$. $H(t)$ is the Heaviside function, defined here as:

$$H(t) = \begin{cases} 0 & \text{if } t < 0 \\ 1 & \text{if } t \geq 0 \end{cases}.$$

If the task has no mandatory part, the first case of (2) is not applicable. It is easy to see that without any loss of generality, it is sufficient to consider integer rewards and time quantums that are natural numbers, i.e., $\forall j, r_{i,j} \in \mathbb{Z}$ and $\forall j, t_{i,j} \in \mathbb{N}$.

It seems that R.1 – R.4 makes the IRIS problem harder to describe and to tackle. However, this is not the case. Respecting R.1 and R.2 allow us to model an IRIS problem with the help of time-dependent Petri nets. (Actually, there exist several classes of Fluid Petri nets, that allows to model continuous reward functions. However, this approach would hardly reduce the problem's complexity.) In turn, Petri nets allow to model dependencies and sharing of resources, as R.3 requests.

Using Petri nets to model the problem is–of course–a detour: Assuming R.1 and R.2 allows us already to describe the IRIS problem as a dynamic programming problem. However, using Petri nets provides some advantages:

- Petri nets are an easy and well-known method, including both system designers and system analyzers.
- There exists a wide range of methods that can be applied to Petri nets, whose correctness is proven; thus additional correctness proofs may be skipped.

3. Timed Petri nets

Petri nets, Petri (1962), are a method to model non-deterministic dynamic systems with discrete states. Petri nets are frequently used to describe scheduling problems, e.g., deadlock recognition.

The original Petri nets do not include a notion of time, which is needed for real-time scheduling problems. There exist a number of extensions to Petri nets

to enable them to deal with time. The most famous approaches are Time Petri nets (Merlin, 1974) and Timed Petri nets (Ramchandani, 1974). In this paper, we use the latter one.

DEFINITION 3.1 (Timed Petri net (TPN))
The structure $Z = (P, T, F, V, D, m, h, m_0, h_0)$ *is called a Timed Petri net (PN) iff*

1. P, T, F *are finite sets with* $P \cap T = \emptyset$, $P \cup T \neq \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ *and* $dom(F) \cup cod(F) = P \cup T$, *where the elements of* P *are called* places, *the elements of* T *are called* transitions, *and the elements of* F *are called* arcs.
2. $V : F \to \mathbb{N}$ *(weight of the arcs)*
3. $D : T \to \mathbb{Q}$ *(duration function) so that* $D(\tau)$ *denotes the delay of the transition* τ.
4. $m : P \to \mathbb{N}$ *(marking)*
5. $h : T \to \mathbb{Q}$ *(transition clock vector)*
6. $m_0 : P \to \mathbb{N}$ *(initial marking)*
7. $h_0 : T \to \mathbb{Q}$ *(initial clock vector marking).*

It is easy to see that considering TPN with $D : T \to \mathbb{N}$ will not result in a loss of generality. Thus, only such duration functions (and accordingly, clock vectors with $h, h_0 : T \to \mathbb{N}$) will be considered in the rest of the paper.

The *marking* of a TPN is a function $m : P \to \mathbb{N}$, such that $m(p)$ denotes a number of *tokens* at the place $p \in P$. To any transition $\tau \in T$ belongs a pre-set $\bullet\tau$ and a post-set $\tau\bullet$, that are given as $\bullet\tau = \{p \mid p \in P \wedge (p, \tau) \in F\}$, and $\tau\bullet = \{p \mid p \in P \wedge (\tau, p) \in F\}$, respectively. Each transition $\tau \in T$ induces the markings τ^+ and τ^-, which are defined as follows: $\tau^- = V(p, \tau)$ and $\tau^+ = V(\tau, p)$. Here, an arc that is not an element of F is assumed to have zero weight.

With the progress of time, the clock vector of a TPN changes. When a time Δt elapses, each clock h_i of the vector h is changed in the following way:

DEFINITION 3.2 (time elapsing) *Given a TPN* Z *with a transition clock vector* h. *When a time* Δt *elapses, each clock* h_i *of the vector* h *is changed in the following way:*

$$h'_i = \max(h_i - \Delta t, 0).$$

A transition τ is *enabled* at marking m iff $\tau^- \leq m$ (i.e., $\tau^-(p) \leq m(p)$ for all places $p \in P$).

DEFINITION 3.3 (maximal step) *Let* Z *be an TPN.* $\mathcal{B} \subset T$ *is called a maximal step at the marking* m *with the transition clock vector* h *iff*

1. $\sum_{\tau \in \mathcal{B}} \tau^- \leq m$
2. $\forall \tau (\tau \in \mathcal{B} \Rightarrow h(\tau) = 0)$
3. $\neg \exists \mathcal{B}^* \left((\mathcal{B}^* \supset \mathcal{B}) \wedge (\mathcal{B}$ *satisfies 1. and 2.*$) \right)$

If at least one enabled transition exists, transitions of the TPN must *fire*. This is a difference with respect to classic Petri nets: there, the net *may* fire, if there is at least one enabled transition. Only maximal steps may fire in a TPN. If there is more than one maximal step that may fire, one of them is selected arbitrarily.

DEFINITION 3.4 (firing) *A TPN Z with the marking m and with a maximal step $\mathcal{B}$ that becomes enabled at time t will change its state the following way:*

1. *At time t:*
   - $\forall \tau \in \mathcal{B}, p \in {}^\bullet\tau, m'(p) = m(p) - \tau^-$
   - $\forall \tau \in \mathcal{B}, h'(\tau) = D(\tau)$
2. *at each time, a clock $h(\tau \in \mathcal{B})$ changes from a non-zero value to zero:*
   - $\forall p \in \tau^\bullet, m'(p) = m(p) + \tau^+$.

Frequently, one finds in the literature the firing to happen in zero time: the tokens of the transition pre-set disappear in the same instance the new token appears at the post-set. However, since the decision about conflicts is made when a transitions becomes enabled, both approaches do not really differ.

An advantage of using Petri nets to model systems is that there exist graphical representations for nets. For TPN, places are represented by circles, transitions are represented by rectangles, where a number denotes the delay, and arcs are represented by arrows, where a number denotes the weight. An actual marking of a Petri net is represented by small filled circles (tokens) inside the marked places.

4. Modeling

In this section, we are discussing different elements of the IRIS problem as described in Section 2. Together, these elements allow for modeling all IRIS problems that meet R.1–R.4.

4.1. A simple task

Let us consider a simple, non-interruptable task with a binary reward function:

$$f_i(t) = \begin{cases} 0 & \text{if } t < t_1 \\ a_i & \text{if } t \geq t_1 \end{cases}. \tag{3}$$

Using the alternative form of (3), the parameters are: $n_i = 1$, $r_{i,1} = a_i$, $t_{i,1} = t_1$. Please, keep in mind that time t here is not the general (wall clock) time, but the fraction of time the task is actually executed. Fig. 1 shows a model of such a simple task.

There, the actual execution is represented by Transition τ_1. A token at places p_1 and p_3 denotes different states of the task, more precisely, "ready to run" and "finished", respectively.

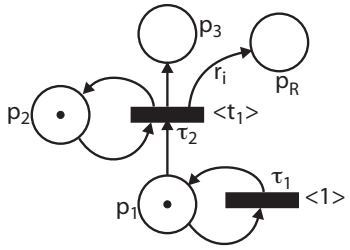

Figure 1. A simple task. Transition delays are marked by sharp brackets.

Place p_2 models a resource needed for execution, e.g., CPUs. Here, the tokens have a different function than at p_1 and p_3: the number of tokens represent the number of resource instances that are available. Finally, the number of tokens at p_R show the reward gained.

A TPN with an enabled transition is forced to fire, but we want to allow an arbitrary delay for the task starts, after they became ready. For this reason, transition τ_1 is added, releasing t_2 from the force to fire instantaneously.

To model a general task with a reward function with the form of (2) but without mandantory part (for tasks with mandantory parts, i.e., rewards of $-\infty$, see discussion in Section 4.3) it is sufficient to compose the more complicated task out of a number of simple tasks, as shown in Fig. 2.

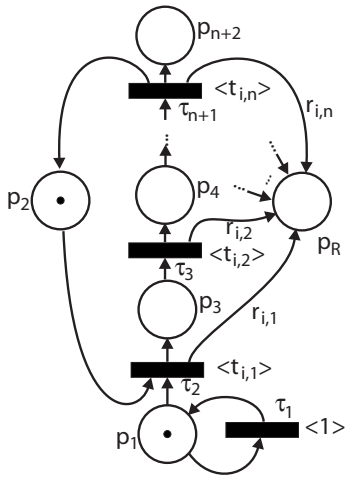

Figure 2. Non-interruptable IRIS task with reward function according to (2) without a mandantory part.

4.2. Interruptable task

To model an interruptable task it would be sufficient to combine a number of simple task elements from Fig. 1 directly, where every computation step represents the minimal computation quantum according R.1. Unlike as shown in Fig. 2, every computation stage returns the consumed resource token (especially the CPU).

However, if all (or at least a number of successive) quantum computation stages gain the same reward (e.g., none), there is a more effective way to model, as shown in the example of Fig. 3.

Firing Transition τ_3 represents the execution of a computation time quantum. The number of such quantums is given by the weight t_1 of the arc (τ_2, p_3). At the beginning of the execution, i.e., after firing of τ_2, place p_3 contains t_1 tokens which are processed step by step. Thus, in this case, a token represents an execution-time quantum. We used this quantum time model already in non-IRIS scheduling, compare, e.g., Richling, Popova-Zeugmann and Werner (2002).

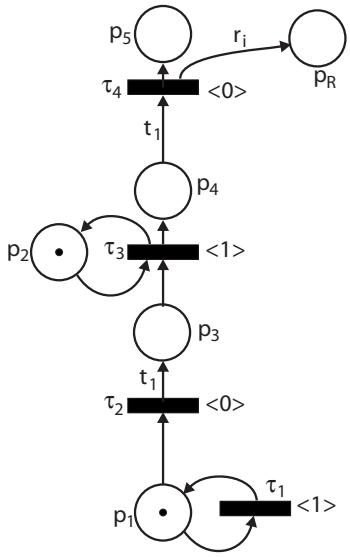

Figure 3. Tokens represent time quantums.

4.3. Rewards

Modeling the reward is straightforward, as already shown in Section 4.1. Every system to model includes exactly one place p_R that displays the reward: the

marking, or more precisely, the number of tokens at a certain time corresponds to the reward gained until this time.

Since firing of transitions models executions of tasks or task parts, changes in reward are assigned to these transitions. I.e., for each transition τ that produces a reward r, $r > 0$, there exists an arc (τ, p_R) with $V(\tau, p_R) = r$. In turn, for each transition τ that produces a reward r, $r < 0$, there exists an arc (p_R, τ) with $V(p_R, \tau) = -r$.

If there exists at least one reward $r_i < 0$, it could be possible that the execution of an action with a negative reward is blocked in the model by an empty reward place, which does not follow the model semantic. To avoid a blocking, the reward place is modelled with an initial marking.

For a system with an overall deadline D_o and where T_n is the set of transactions that are negatively rewarded, $\tau_{n_i} \in T_n$, the following formula gives an upper bound for the initial marking of p_R:

$$m_0(p_R) = \sum_{\tau_{n_i} \in T_n} \left\lceil \frac{D_0}{D(\tau_{n_i})} \right\rceil \cdot V(p_R, \tau_{n_i}). \tag{4}$$

It is not necessary (and feasible) to model the $-\infty$ reward. If tasks have mandatory parts, it can be modeled by rewards and side conditions for the optimization problem. Each mandatory part has to provide a reward value that is large enough to not be compensated by execution of optional parts prior to the deadline. Let T_m be the set of transitions that represent mandatory parts and $T_p = \{v_i\}$ the set of transitions with $V(\tau_i, p_R) > 0$ (note that a mandatory part never has a negative reward, i.e., $T_m \cap T_n = \emptyset$).

The following formula gives an upper bound for the rewards of part in T_p:

$$r_m = \sum_{\tau_i \in T_p} \left\lceil \frac{D_0}{D(\tau_i)} \right\rceil \cdot V(\tau_i, p_R). \tag{5}$$

Thus, for the transitions that represent mandantory parts, the following must hold:

$$\forall \tau \in T_m, V(\tau, p_R) \geq r_m. \tag{6}$$

In addition, only solutions of the IRIS problem will be accepted where the total reward r_t meets the the following inequality:

$$r_t \geq \sum_{\tau \in T_m} V(\tau, p_R) + m_0(p_R). \tag{7}$$

Note that only transitions τ with a delay $D(\tau) > 0$ contribute to changes in the reward. In addition, there are no two or more places, that form a ring with zero-delay transitions. Thus, the state graph (see Section 5) will not include any loops.

4.4. Putting together

The elements introduced in this section allow for modeling arbitrary Petri nets according to the IRIS problem from Section 2.2. An arbitrary task with a reward function of the form of (2) is simply constructed of a sequence of simple task, which are constructed in either of the ways seen in Figs. 1–3.

Dependencies among tasks can be modeled by Petri net pre-conditions, i.e., drawing arcs from the "ready" place of a predecessor task to the first action (transition) of the successor task. Other dependencies, i.e., shared ressources, can be handled in the same way as CPUs.

5. Path problem

After constructing the model, the problem of an optimal IRIS schedule can now be mapped to a critical path problem (also known as longest path problem), which is solvable with the known method of dynamic programming, see, e.g., Cormen, Leiserson, Rivest (1990).

The straightforward solution is to spawn the timeless Petri net state space, seek the state with the highest $m(p_r)$ that meets the side conditions, and finally use DP to find the path to this state and check, whether this path meets the deadline. A similar approach is followed, e.g., in Popova-Zeugmann, Werner (2005). However, the state space of IRIS task models constructed as described in Section 4 may be infinite (although the Petri net itself is finite) since infinite increases of rewards are allowed. And even if the state space is finite, frequently many of the found states will not meet the deadline constraint. (Considering found states only does not allow for checking the side condition: Timed Petri net states do not include global time).

We suggest to use a modified state space that is augmented by a global time but reduced by the marking of p_r. Instead, gain and loss of reward serve as weights in the state graph. Then, the relaxing operation of a path algorithm can be applied to $m(p_r)$.

By removing p_r from the Petri net, it becomes bounded, and thus, its state space becames finite. However, adding an absolute time, makes the state space infinite, again. But not all of this state space is of interest. Only states with an absolute time smaller than the deadline have to be considered.

Our approach is to spawn the state space graph (or, more precisely: a part of it) and to solve the critical path problem at the same time. The modified state space is developed from the initial state. The initial modified state S'_0 consists of the initial timeless Petri net state without p_r, i.e., m_0^-, augmented with the absolute initial time $t_a = 0$. I.e., $S'_0 = (m_0^-, 0)$.

To calculate future marking, the state equation from Popova-Zeugmenn, Werner and Richling (2003) may be used. Here, the marking is not a vector, but a matrix of the dimension $|P| \times d_{max}$ (timed marking) where d_{max} is the largest delay in the Petri net plus one. C is the corresponding incidence matrix,

Ψ the Parikh matrix describing the (sequence of) step(s), and R a progress matrix of the form

$$R = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & & \vdots & \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}.$$

The (timed) marking after Δt time units is calculated by:

$$m_{i+\Delta t} = m_i \cdot R^{\Delta t} + C \cdot \Psi. \tag{8}$$

The time component increases simply with time:

$$t_{a_{i+\Delta t}} = t_{a_i} + \Delta t. \tag{9}$$

States in the modified state space are identical, iff the reduced Petri net state is the same and the absolute time is the same. The arcs of the state graph are labeled with the changes of m_{p_r}.

To increase efficiency, it is possible to execute parts of the critical path algorithm (initialization and first round of relaxing) together with spawning of the state space. After the termination of the critical path algorithm, all path weights to the states with $t = D_o$ are known as well as the critical path that corresponds with the highest reward.

The complexity of this algorithm is polynomial with the number of states in the modified state space. This number, however, may be exponential in the worst case, bounded by $|T|^{\frac{D}{t_{max}}}$. Yet, in many realistic cases, this number is polynomial to $|T| \cdot |P|$.

6. An example

As an example consider the following simple system: two IRIS task J_1, J_2 have to share one processor, and J_2 depends on the result of J_1. The reward functions of each task are given in the form of sets of $(t_{i,j}, r_{i,j})$:

J_1: $\{(t_a, r_a), (2t_a, r_a), (3t_a, r_a), (4t_a, r_a)\}$
J_2: $\{(t_a, 0), (4t_a, 3r_a)\}$.

Fig. 4 shows the graphs for these reward functions. The problem is oversimplified: One can easily see that, if the deadline is at least $4t_a$, any execution is optimal that includes both parts of J_2, the mandatory part of J_1, and as many as possible of the optional parts of J_1. If the deadline is shorter than $4t_a$, but larger than $2t_a$, any execution is optimal that includes both mandatory parts and as many as possible optional parts of J_1. For a deadline shorter than $2t_a$, no solution exists. For the example, let us assume a deadline of $5t_a$.

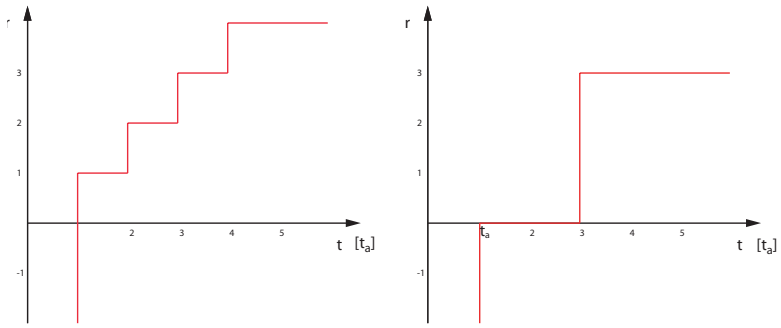

Figure 4. Reward functions for example system

The corresponding Petri Net is presented in Fig. 5. Please note that the arc (τ_2, P_2) has a multiplicity of 3, allowing to fold the three optional parts of J_1. Place P_3 belongs to both, J_1 and J_2: for J_1 a marking signals that a result is reached, and for J_2 it marks the task as ready to run. It is sufficient to set the rewards of the mandatory parts to $6r_a$. Thus, no solution is accepted, if the gained reward is not at least $12r_a$.

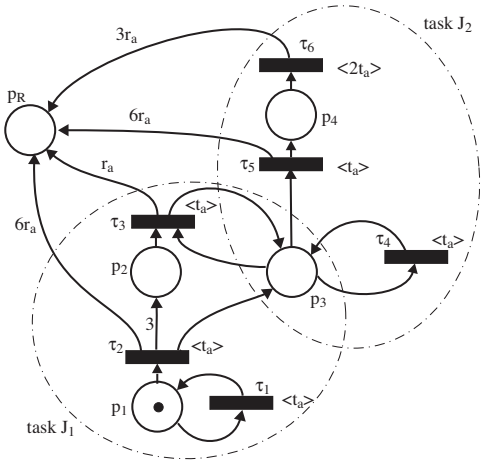

Figure 5. Petri net of the example system

Since all delays are multiplicities of t_a, it makes sense to declare t_a as the time unit of the net. The net incidence matrix and initial time marking are as

follows:

$$C = \begin{pmatrix} -1\,1\,0 & -1\,0\,0 & 0\,0\,0 & 0\,0\,0 & 0\,0\,0 & 0\,0\,0 \\ 0\,0\,0 & 0\,3\,0 & -1\,0\,0 & 0\,0\,0 & 0\,0\,0 & 0\,0\,0 \\ 0\,0\,0 & 0\,1\,0 & -1\,1\,0 & -1\,1\,0 & -1\,0\,0 & 0\,0\,0 \\ 0\,0\,0 & 0\,0\,0 & 0\,0\,0 & 0\,0\,0 & 0\,1\,0 & -1\,0\,0 \\ 0\,0\,0 & 0\,6\,0 & 0\,1\,0 & 0\,0\,0 & 0\,1\,0 & 0\,0\,2 \end{pmatrix} \qquad m_0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Fig. 6 shows the modified state graph of the Petri net, where the edges are labled with the marking changes of the reward place (see Section 5). The initial state is denoted by a double border. The spawning of the state graph is cut where a state includes a time greater or equal to the deadline. States with times grater than the deadline (marked with dashed borders) must not be a part of the solution.

To calculate the optimal schedule, a critical path alogrithm will be applied to the modified state graph. There exist quite a number of critical path algorithms. Without loss of generality we use the Bellman-Ford algorithm (Bellman, 1958; Ford and Fulkerson, 1962). The Bellman-Ford algorithm is a shortest path algorithm; however, it is easy to modify it to a critical path algorithm. Denote the state graph as $G = (S, E)$, where S is the set of states $s \in S$, E is the set of directed edges $e_{ij} \in E$, $e_{ij} = (s_i, s_j)$; s_0 is the initial state and $w : E \to \mathbb{Z}$ is the weight function of the edges (i.e. the corresponding marking changes of the reward places). The critical path algorithm is as follows (the original Bellman-Ford algorithm contains a check if there exist negative cycles; however, since our state graph is an acyclic graph, this check is not needed):

for all $s \in S$ **do**
  $d(s) \Leftarrow -\infty$
  $n(s) \Leftarrow nil$
end for
$d(s_0) \Leftarrow 0$
for $i \Leftarrow 1$ **to** $|V| - 1$ **do**
  for all $e_{ij} \in E$ **do**
    if $d(s_j) < d(s_i) - w(e_{i,j})$ **then**
      $d(s_j) \Leftarrow d(s_i) - w(e_{ij})$
      $n(s_2) = s_1$
    end if
  end for
end for

After the algorithm terminates, the path $(s_0, n(s_0), n(n(s_0)), \ldots)$ describes the schedule with the highest reward. In our example, this is the sequence $\{\tau_2, \tau_3, \tau_5, \tau_6\}$, i.e., the execution of task J_1 has to end and task J_2 has to start, after the first

optional part of J_1 is executed, and no idle times are allowed. In Fig. 6, this path is marked with gray edges.

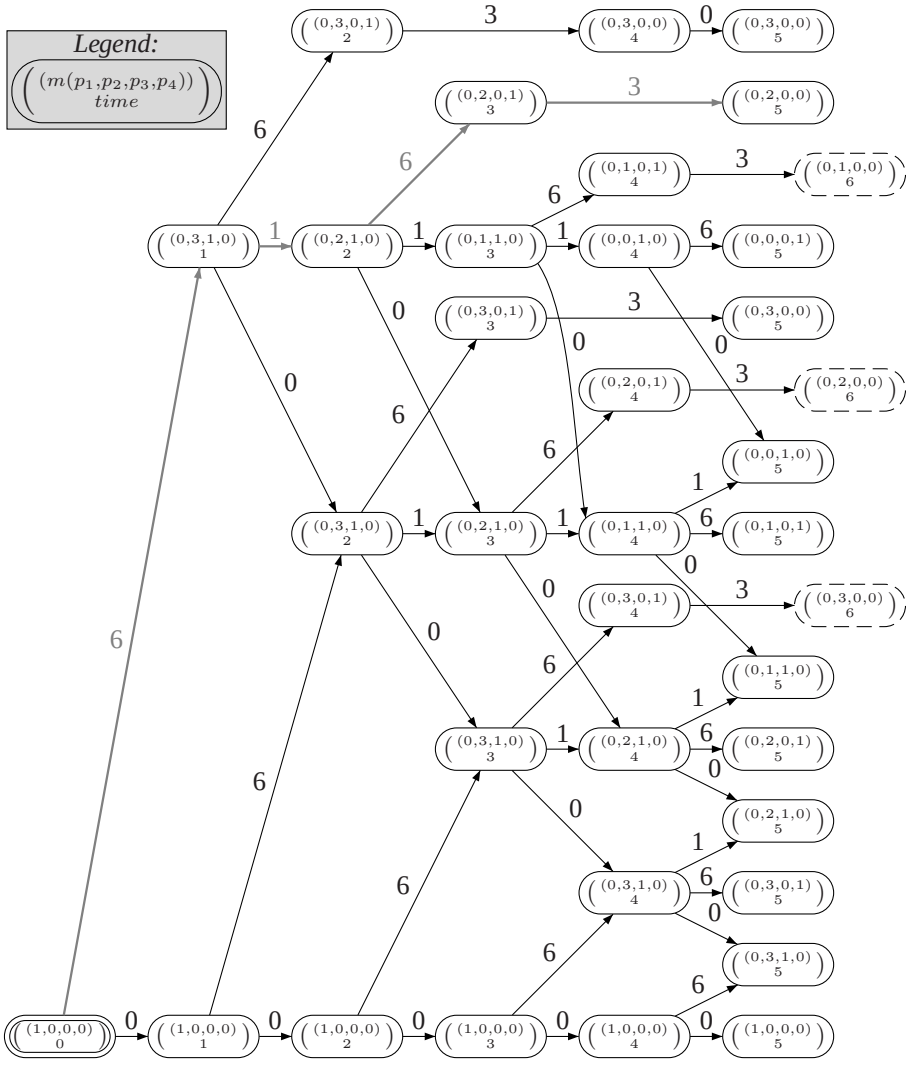

Figure 6. Modified state graph of the example system

7. Conclusion

Within this paper, we stated a number of realistic assumption for IRIS scheduling problems, allowing us to describe such problems with Timed Petri nets. We presented the different components to model these IRIS problems and discussed the finding of an optimal solution with the help of critical path algorithms.

References

Bellman, R. (1958) On a routing problem. *Quarterly of Applied Mathematics* **16** (1), 87–90.

Cormen, T.H., Leiserson, C.E. and Rivest, R.L. (1990) *Introduction to Algorithms*. MIT Press/McGraw-Hill, Cambridge, MA.

Dey, J.K. Kurose, J.F. Towsley, D.F. Krishna, C.M. and Girkar, M. (1993) Efficient on-line processor scheduling for a class of IRIS (increasing reward with increasing service) real-time tasks. In: *Measurement and Modeling of Computer Systems*, 217–228.

Ford, L.R. and Fulkerson, D.R. (1962) *Flows in Networks*. Princeton University Press, Princeton, New Jersey.

Krishna, C.M. and Shin, K.G. (1997) *Real-Time Systems*. McGraw Hill, New York, NY.

Liu, J.W.S., Shih, W.-K., Lin, K.-W., Bettati, R. and Chung, J.-Y. (1994) Imprecise computations. *Proc. of the IEEE*, January, **82** (1), 83–94 .

Liu, J.W.S., Lin, K.-J., Yu, A.C.-S., Chung, J.-Y. and Zhao, W. (1991) Algorithms for scheduling imprecise computations. *IEEE Computer* **24** (5).

Merlin, P. (1974) *A Study of the Recoverability of Communikation Protocols*. PhD thesis, Irvine.

Petri, C.A. (1962) *Kommunikation mit Automaten*. Schriften des IIM **2**, Institut für Instrumentelle Mathematik, Bonn.

Popova-Zeugmann, L. and Werner, M. (2005) Extreme runtimes of schedules modelled by time petri nets. *Fundamenta Informaticae* **67** (1–3), 163–174.

Popova-Zeugmann, L., Werner, M. and Richling, J. (2003) Using state-equation to prove non-reachability in timed petrinets. *Fundamenta Informaticae* **55** (3), 187–202.

Press, W.H. Teukolsky, S.A. Vetterling, W.T. and Flannery, B.P. (1992) *Numerical Recipies in FORTRAN*. Cambridge University Press, Cambridge.

Ramchandani, C. (1974) Analysis of asynchronous concurrent systems by Timed Petri Nets. Project MAC, *Technical Report* **120**, MIT.

Richling, J., Popova-Zeugmann, L. and Werner, M. (2002) Verification of non-functional properties of a composable architecture with petrinets. *Fundamenta Informaticae* **51** (2), 185–200.

Rölke, H. and Heitmann, F. Petri Nets World. Web site. `http://www.informatik.uni-hamburg.de/TGI/PetriNets/bibliographies`.

Shih, W.-K. and Liu, J.W.S. (1995) Algorithms for scheduling imprecise computations with timing constraints to minimize maximum error. *IEEE Trans. on Computers*, **44** (3), 466–471.

Yee, S.T. and Ventura, J.A. 1999 A dynamic programming algorithm to determine optimal assembly sequences using petri nets. *International Journal of Industrial Engineering - Theory, Applications and Practice* **6** (1), 27–37.