Embedded Hard Real-Time Systems Scheduling: An Unmanned Ground Vehicle Case Study

Raimundo Barreto[1], **Marília Neves**[2], **Eduardo Tavares**[2], **and Paulo Maciel**[2]

[1] Departamento de Ciência da Computação - UFAM
Av. Rodrigo Otávio, 3000, Aleixo, 69077-000, Manaus-AM-Brazil

[2]Centro de Informática -UFPE
PO Box 7851, 50732-970, Recife-PE-Brazil

{rsb, mln2, eagt, prmm}@cin.ufpe.br

Abstract. *Finding a hard real-time feasible schedule is not trivial since this problem is NP-hard in its general form. There are two general approaches for scheduling tasks: runtime and pre-runtime scheduling. For many cases, runtime methods do not find a feasible schedule even if such a schedule exists. Such situations often occurs when the design model imposes intertask relations, such as precedence and exclusion relations. The method proposed in this work finds a pre-runtime scheduling, provided that one exists, using state space exploration. The main problem with such methods is the space size, which can grow exponentially. This paper applies minimization methods on the state space, and presents a depth-first search method on a timed labeled transition system derived from the time Petri net model. This model is a compact and precise representation of tasks, their relations and constraints.*

1. Introduction

Embedded hard real-time systems are dedicated computer applications having to satisfy stringent timing constraints, that is, they must guarantee that all tasks complete before their deadlines. A failure to meet deadlines may have serious consequences such as resources damage or even loss of human life. In order to meet this requirement, scheduling performs an important role. Although we have proposed a method for scheduling in multiprocessors architecture [Barreto et al., 2004], in this paper the discussion is restricted to uniprocessor environment.

In real-time systems, there are two general approaches for scheduling tasks: runtime and pre-runtime scheduling. In *runtime scheduling*, schedules are computed on-line as tasks arrive, using a priority-driven approach. However, there are situations where this approach is not able of finding a feasible schedule, even when such schedule exists. On the other hand, in a pre-runtime scheduling algorithm the schedule is computed entirely off-line, which can achieve 100% processor utilization, and it can reduce context-switching. This approach is considered inflexible, but it is not a concern if it is considered embedded systems.

This work uses *state space exploration* since it provides a complete automatic strategy for verifying finite-state systems [Godefroid, 1994]. It consists in recursively checking all successor states, starting in a given initial state, by executing all enabled action in each state. In spite of a feasible schedule can be found using such strategy, it may be limited by the excessive size of its state space. This problem comes up due to the analysis based on the interleaving of concurrent activities. This exponential growth is known as the *state explosion problem* [Godefroid, 1994, Valmari, 1998]. This paper applies minimization methods on the state space, and presents a depth-first search algorithm for finding a feasible schedule on the minimized state space.

The rest of the paper is organized as follows: Section 2 shows related works. Section 3 presents the computational model syntax and semantics. Section 4 describes the task model. Section 5 introduces the formal modeling methodology and Section 6 shows how to synthesize pre-runtime schedules. Section 7 presents experimental results, explaining in more details an unmanned ground vehicle case study. Finally, Section 8 shows some conclusions.

2. Related Work

Xu and Parnas [Xu and Parnas, 1990] present an algorithm that finds an optimal pre-runtime schedule on a single processor for real-time process segments with release, deadlines, and arbitrary exclusion and precedence relations, using a branch-an-bound algorithm. In spite of the importance of this work, it does not presented real-world experimental results. Shepard and Gagné [Shepard and Gagné, 1991] extended the work of Xu and Parnas by proposing an implicit enumeration technique for dealing with multiprocessors, but as pointed out in [Abdelzaher and Shin, 1997], the algorithm occasionally fails in finding existing feasible schedules, since it attempts to reduce schedule lateness by modifying only the schedule of the processor running the latest segment. Abdelzaher and Shin [Abdelzaher and Shin, 1999] proposed an extension to the Xu and Parnas' pre-run-time scheduling algorithm in order to deal with distributed real-time systems. This algorithm takes into account delays, precedence relations imposed by interprocess communications, and considers many possibilities for improving the scheduling lateness at the cost of complexity.

Several authors also use Petri nets in scheduling theory (e.g. [Xu et al., 2002]). However, they are only concerned with schedulability analysis, relying on a well-known priority policy, which may not find feasible schedules if arbitrary precedence and exclusion relations are considered.

Comparing the proposed approach with previous works, it differs in the sense that: (1) Previous works model the *scheduling problem*, not the *system*. This work uses a time Petri net formalism for system's modeling in order to find a feasible scheduling. Furthermore, the model can also be used to synthesize predictable and timely scheduled code; and (2) Using Petri net analysis techniques allows one to check several system properties, such as, reachability, deadlock-freedom, boundedness, fairness, etc.

Although state space exploration is not new, at the best of our present knowledge, there is no similar work that uses formal methods for modeling real-time systems with the aim of finding a feasible pre-runtime scheduling. This work brings an important contribution since it opens up a new possibility for analyzing a problem studied for many years.

3. Computational Model

The computational model syntax is given by a time Petri net [Merlin and Faber, 1976], which is a Petri net [Murata, 1989] extended with time, and its semantics is given by its timed labeled transition system.

A time Petri net (TPN) is a bipartite directed graph represented by a tuple $TPN = (P, T, F, W, m_0, I)$. P (ordered set of places), and T (ordered set of transitions) are non-empty disjoint sets and represent the two types of nodes in the graph. The edges are represented by $F \subseteq (P \times T) \cup (T \times P)$, which is a flow relation. $W : F \rightarrow \mathbb{N}$ represents the weight of the flow relation (F). A TPN marking m_i is a vector $m_i \in \mathbb{N}^{|P|}$, and m_0 is the initial marking. Finally, $I : T \rightarrow \mathbb{N} \times \mathbb{N}$, represents the timing constraints, where $I(t) = (EFT(t), LFT(t))$ $\forall t \in T$ and $EFT(t) \leq LFT(t)$. The lower and upper bound are called earliest and latest firing time, respectively.

A set of enabled transitions is denoted by: $ET(m_i) = \{t \in T \mid m_i(p_j) \geq W(p_j, t)\}$, $\forall p_j \in P$. $C \in \mathbb{N}^{|ET(M)|}$ is a clock vector, which represents the time elapsed since the respective transition enabling. In order to facilitate the TPN's analysis, it is important to differentiate static and dynamic intervals associated with transitions. The dynamic firing interval ($I_D(t)$) is composed by a dynamic lower bound (DLB), and a dynamic upper bound (DUB). I_D is computed as follows: $DLB(t) = max(0, EFT(t) - c(t))$, and $DUB(t) = LFT(t) - c(t))$. As it can be seen,

$I_D(t)$ is dynamically modified whenever the respective clock variable is incremented, and t does not fire.

The set of states of a TPN is given by $S \subseteq (M \times \mathbb{N}^{|ET(M)|})$, that is, a single state is defined by a pair (m, c), where m is a marking, and c is its respective clock vector for $ET(m)$. The initial state of a TPN is $s_0 = (m_0, c_0)$, where $c_0(t) = 0\ \forall t \in ET(m_0)$. $FT(s)$ is the set of firable transitions at state s defined by: $FT(s) = \{t_i \in ET(m) | DLB(t_i) \leq min(DUB(t_k)) \forall t_k \in ET(m)\}$, where $FT \subseteq ET \subseteq T$. The *firing domain* for t at a specific state s, is defined by: $FD_s(t) = [DLB(t), \min(DUB(t_k))]$, $\forall t_k \in ET(m)$.

The semantic of a time Petri net $\mathcal{P} = (P, T, F, W, M_0, I)$ is defined by associating a timed labeled transition system (TLTS) $\mathcal{L}_{\mathcal{P}} = (S, \Sigma, \rightarrow, s_0)$ such that: (i) S is the set of states of $\mathcal{P}$; (ii) $\Sigma \subseteq (T \times \mathbb{N})$ is a set of activities labeled with (t, θ) corresponding to the firing of a firable transition at a specific time value in the firing interval $FD(s)$, $\forall s \in S$; (iii) $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation; and (iv) s_0 is the initial state of $\mathcal{P}$.

For a state transition $\langle s, (t, \theta), s' \rangle$ in $\rightarrow$, it is denoted by $s \xrightarrow{(t,\theta)} s'$, implying that the system can change its state from s to s' on activity (t, θ). Let $\mathcal{L}_{\mathcal{P}}$ be a TLTS of a TPN $\mathcal{P}$, where s_0 its initial state, $s_n = (m_n, c_n)$ a final state, and $m_n = M^F$, which is the desired final marking. $s_0 \xrightarrow{(t_1,\theta_1)} s_1 \xrightarrow{(t_2,\theta_2)} s_2 - - \rightarrow s_{n-1} \xrightarrow{(t_n,\theta_n)} s_n$ is defined as a *feasible firing schedule*, where s_i = $\texttt{fire}(s_{i-1}, (t_i, \theta_i))$, $i > 0$, if $t_i \in FT(s_{i-1})$, and $\theta_i \in FD_{s_{i-1}}(t_i)$. As it is presented later, the modeling methodology guarantees the final marking M^F is well-known since it is explicitly modeled.

4. Task Model

The *task model* is composed by: (i) a set of periodic preemptable tasks with bounded discrete time constraints; and (ii) intertask relations, such as precedence and exclusion relations. Let $\mathcal{T}$ be the set of tasks in a system.

Definition 4.1 (Periodic Task) *Let τ_i be a periodic task defined by $\tau_i = (ph_i, r_i, c_i, d_i, p_i)$, where ph_i is the initial phase; r_i is the release time; c_i is the worst case computation time; d_i is the deadline; and p_i is the period.*

A periodic task samples objects of interest at a fixed rate. The phase (ph_i) is the delay associated to the first time request of task τ_i after the system starting. $ph_i = 0$ whenever not specified. The period in which τ_i is requested is denoted by p_i, and c_i is the worst case computation time required for executing task τ_i. Release time r_i, and deadline d_i, are time instants related to the beginning of a period. Thus, r_i is the earliest time where task τ_i may start execution; and d_i is the time at which task τ_i must be completed. This work considers $c_i \leq d_i \leq p_i$. The definition of the initial phase is important, since non schedulable system may become schedulable when an initial phase is specified. For instance, considering two tasks, τ_1 and τ_2, having equal timing constraints $(ph_1, r_1, c_1, d_1, p_1) = (ph_2, r_2, c_2, d_2, p_2) = (0, 0, 5, 5, 10)$. As it can be seen, this system is not schedulable. However, if an initial phase is specified, i.e. $ph_2 = 5$, the system becomes schedulable.

Definition 4.2 (Sporadic Task) *Let τ_k = (c_k, d_k, min_k) be a sporadic task, where c_k is the worst case computation time; d_k is the deadline; and min_k is the minimum period between two activations of task τ_k.*

A task is classified as sporadic if it can be randomly activated, but the minimum period between two activations is known. As pre-runtime approaches may only schedule periodic tasks, the sporadic tasks have to be translated to an equivalent periodic task. Based on the Mok's work [Mok, 1983], one technique was derived for tackling such problem where each sporadic task (c_s, d_s, min_s) is translated into a corresponding periodic task $(ph_p, r_p, c_p, d_p, p_p)$, satisfying the following conditions: $ph_p = r_p = 0$, $c_p = c_s$, $\quad d_s \geq d_p \geq c_s$, $\quad c_s \leq p_p \leq \min(d_s - d_p + 1, min_s)$. For example, consider a sporadic task defined by $ph_s = 0$; c_s = 2;

d_s= 9; and min_s = 10. The corresponding periodic process can be: $(0, 0, 2, 2, 8)$, where $ph_p = 0$, $r_p = 0$, $c_p = c_s = 2$, $d_p = c_s = 2$, and $p_p = min(d_s - d_p + 1, min_s) = min(8, 10) = 8$. In this case, the periodic executions are scheduled to start at time 0, 8, 16, ..., and if the sporadic request are, for instance, 1, 11, and 30, then the start times of the sporadic tasks executions are 8, 16, and 32. As it can be noted, despite sporadic tasks arriving happen at random, they can be dealt with as periodic ones by buffering of such events, and this translation makes d_s to be always met. Figure 1 shows graphically how is the behavior of the equivalent periodic tasks related to the sporadic requests. In that figure, rs_i's are sporadic requests, and s_i's are actual sporadic executions. As it can be seen, all timing constraints for sporadic tasks are satisfied by the equivalent periodic task.

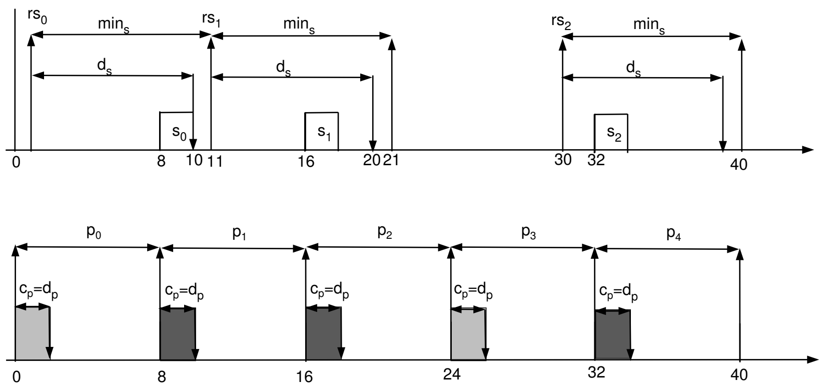


Figure 1: Sporadic constraints are satisfied

A task τ_i *precedes* task τ_j, if τ_j can only start executing after τ_i has finished. In general, this kind of relation is suitable whenever a task (successor) needs information that is produced by another task (predecessor). A task τ_i *excludes* task τ_j, if no execution of τ_j can start while task τ_i is executing. If it is considered a single processor, then task τ_i could not be preempted by task τ_j. Exclusion relations may prevent simultaneous access to shared resources. In this work it is considered that the exclusion relation is not symmetrical, that is, when A `EXCLUDES` B it does not necessarily implies that B `EXCLUDES` A.

Each task $\tau_i \in \mathcal{T}$ consists of a finite sequence of *task time units* $\tau_i^0, \tau_i^1, \cdots, \tau_i^{c_i-1}$, where τ_i^{j-1} always precedes τ_i^j, for $j > 0$. A task time unit is the smallest indivisible granule of a task, during which it cannot be preempted by any other task. It is worth noting that the total number of task time units is equal to the computation time required by that task. A task can also be split into more than one *subtasks*, where each subtask is composed by one or more task time units.

5. Modeling Real-Time Systems

Hard real-time systems are those that besides its functional correctness, timeliness must be satisfied. The modeling phase is very important to attain such constraints.

5.1. Scheduling Period

The proposed method schedules the set of periodic tasks occurring in a period that is equal to the least common multiple (LCM) of the periods of the given set of tasks. The LCM is also called *schedule period* (P_S). Within this new period, there are several *tasks instances* of the same task, where $N(t_i) = P_S/p_i$ gives the instances of t_i. For example, consider the following task model consisting of two tasks: $t_1 = (0, 0, 2, 7, 8)$ and $t_2 = (0, 2, 3, 6, 6)$. In this particular case, $P_S = 24$, implying that the two periodic tasks are replaced by seven new periodic tasks ($N(t_1) = 3$, and $N(t_2) = 4$), where the timing constraints of each task instance has to be transformed to consider that new period [Xu and Parnas, 1990].

5.2. Scheduling Methods

Figure 2 presents three ways for modeling scheduling methods, where $c = cs_1 + cs_2$ is the task computation time (cs_1 and cs_2 are computation times for the first and last subtask, respectively):

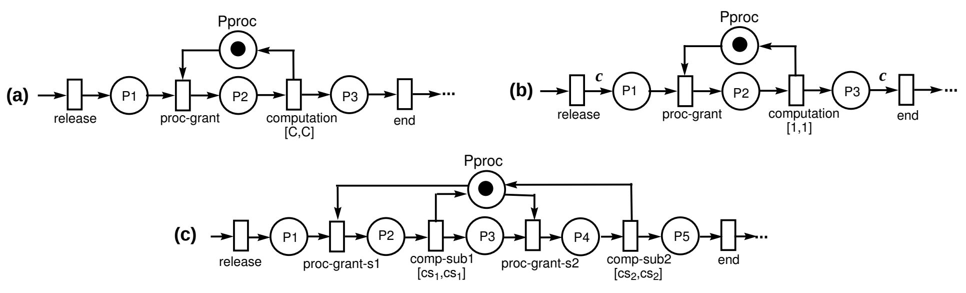


Figure 2: Modeling Scheduling Methods

a) *all-non-preemptive*: processor is just released after the entire computation be finished. Figure 2(a) shows that computation transition timing interval has bounds equal to the task computation time (i.e., [*c*, *c*]);
b) *all-preemptive*: tasks are implicitly split into all possible subtasks. This method allows running other *conflicting tasks*, meaning that one task could preempt another task. It is worth observing, the difference between the timing interval for the computation transition and the arc weight in Figures 2(a) and 2(b).
c) *defined subtasks*: tasks are split into more than one explicitly defined subtasks. Figure 2(c) shows two subtasks.

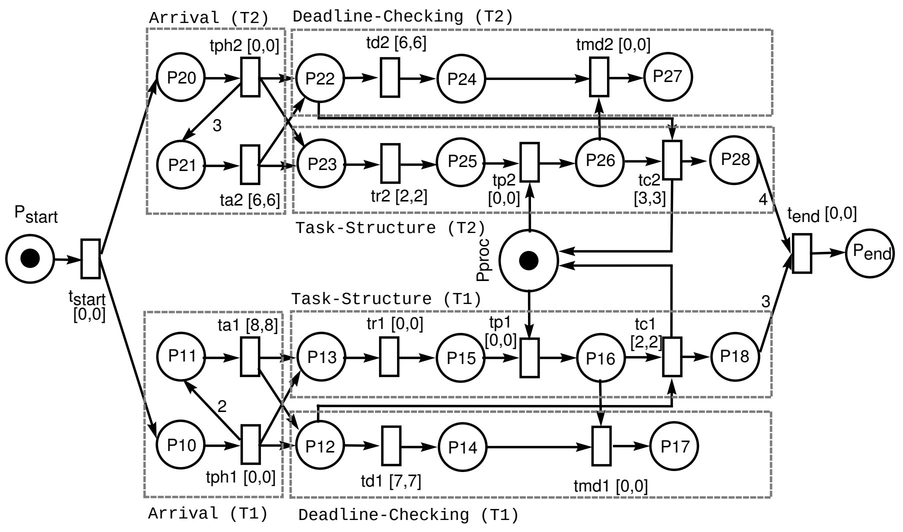


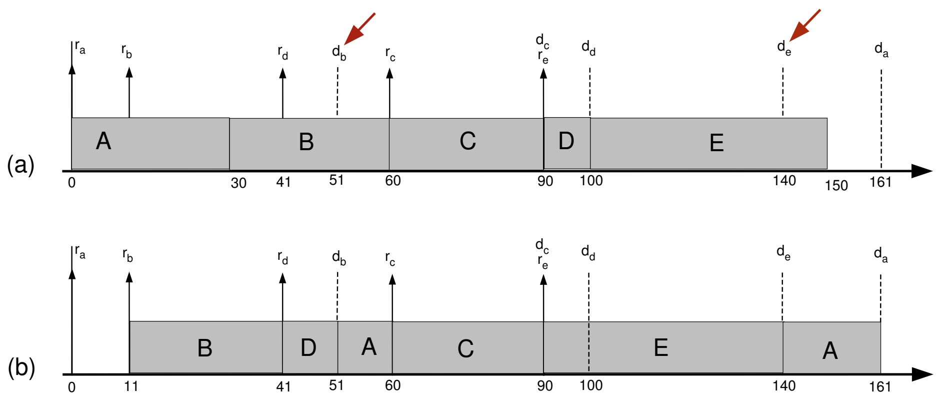
Figure 3: Petri net model

5.3. Tasks Modeling

Figure 3 is used to show (in dashed boxes) the three main *building blocks* for modeling a real-time task. These blocks are: *(a) Task Arrival*, which models the periodic invocation for all task's instances. Transition t_{ph} models the initial phase, whilst transition t_a models the periodic arrival for the remaining instances; *(b) Deadline Checking*, where it is used elementary net structures to capture deadline missing. Some works (e.g. [Altisen et al., 1999]) extended the Petri net model for dealing with deadline checking. *(c) Task Structure*, which models: release time, processor granting, computation, and processor releasing. In Figure 3 it is presented a non-preemptive TPN model for the example presented in previous subsection. It does not model the seven task instances. Instead, it models only the two original tasks, and the time period of every task instances.

6. Pre-Runtime Scheduling Synthesis

This section investigates how to find a feasible pre-runtime schedule using state space exploration. First of all, it presents a brief comparison among runtime and pre-runtime approaches. Later, it describes how to minimize the state space size, and finally, it presents an algorithm that implements the proposed method.



D

Figure 4: Comparison between Runtime and Pre-runtime Scheduling

6.1. Runtime versus Pre-runtime

The runtime approach (static or dynamic priority based scheduling) may constrain the possibility of finding a feasible schedule, even if such schedule exists. For instance, consider the specification model consisting of five tasks, A, B, C, D, E, and the respective timing constraints: $A = (0, 30, 161)$; $B = (11, 30, 51)$; $C = (60, 10, 90)$; $D = (41, 10, 100)$; and $E = (90, 50, 140)$. This specification also considers that B PRECEDES D, A EXCLUDES B, and A EXCLUDES D. Figure 4(a) shows that a runtime approach could not find a feasible schedule, since tasks B and E miss their deadlines. However, a pre-runtime approach finds a feasible schedule (Figure 4(b)). In this case, processor must be left idle between time 0 and 11, even though A's release time is 0, since if A starts, it would cause B and E to miss their deadlines.

In order to provide predictability in a complex hard real-time system, the major characteristics of tasks must be known (or bounded) in advance, otherwise it would be impossible to guarantee a priori that all timing constraints will be met. In accordance with [Xu and Parnas, 1993], pre-runtime scheduling is often the only means of providing predictability in complex systems.

6.2. Minimizing State Space Size

6.2.1. Partial-Order Reduction.

When generating TLTS of a time Petri net (or even of process algebra) tasks' interleaving is the fundamental point to be considered when analyzing state space explosion problem. As example, the analysis of n concurrent activities needs searching all $n!$ interleaving possibilities. If activities can be executed in any order, such that the system always reaches the same state, these activities are *independent*. In other words, it does not matter in which order the activities are executed. The partial-order reduction method explores the independence of activities [Godefroid, 1994].

The independent activities are those that do not disable any other activity, such as: arrival, release, precedence, computation (after the processor granting), processor releasing, and end-task. This reduction method proposes giving to each class of independent activities different *choice priority* levels. The other activities, the dependent ones, like *exclusion* and *processor granting*, have lowest choice priority. Therefore, when changing from one state to another state, it is sufficient to analyze the class with highest choice priority and pruning the other ones. When all independent activities are executed, certainly the final state is the same, because the order between them does not matter.

This reduction is important due to two reasons: (i) depleting the amount of storage; and (ii) finding a fast negative result, when the system does not have a feasible schedule.

6.2.2. Removing Undesirable States.

In Section 5 it is presented how to model undesirable error states (or markings), for instance, states that represent missed deadlines. The method proposed is of interest for schedules that do not reach any of these undesirable states. For this reason, when generating the TLTS, the transitions leading to undesirable error states, represented by the set T^E (which is well-known for the designer) have to be discarded.

Table 1: Illustrative example

#	st	ET	C	PT	trans+time
1	0	{tstart}	{0}	{tstart}	{tstart,0}
2	1	{tph1,tph2}	{0,0}	{tph1,tph2}	{tph1,0}
3	2	{tph2,tr1,ta1,td1}	{0,0,0,0}	{tph2}	{tph2,0}
4	3	{tr1,ta1,ta2,td1,td2}	{0,0,0,0,0}	{tr1}	{tr1,0}
5	4	{tp1,ta1,ta2,td1,td2}	{0,0,0,0,0}	{tp1}	{tp1,0}
6	5	{tr2,tc1,ta1,ta2,td1,td2}	{0,0,0,0,0,0}	{tr2}	{tr2,2}
7	6	{tc1,ta1,ta2,td1,td2}	{2,2,2,2,2}	{tc1}	{tc1,0}
8	7	{tp2,ta1,ta2,td2}	{0,2,2,2}	{tp2}	{tp2,0}
9	8	{tc2,ta1,ta2,td2}	{0,2,2,2}	{tc2}	{tc2,3}
10	9	{ta1,ta2}	{5,5}	{ta2}	{ta2,1}
11	10	{ta1,ta2,tr2,td2,}	{6,0,0,0}	{ta1}	{ta1,2}
12	11	{ta1,ta2,tr1,tr2,td1,td2}	{0,2,0,2,0,2}	{tr1,tr2}	{tr1,0}
13	12	{ta1,ta2,tr2,td1,td2,tp1}	{0,2,2,0,2,0}	{tr2}	{tr2,0}
14	**13**	**{ta1,ta2,td1,td2,tp1,tp2}**	**{0,2,0,2,0,0}**	**{tp1,tp2}**	**{tp1,0}**
15	14	{ta1,ta2,td1,td2,tc1}	{0,2,0,2,0}	{tc1}	{tc1,2}
16	15	{ta1,ta2,td2,tp2}	{2,4,4,0}	{tp2}	{tp2,0}
17	16	{ta1,ta2,td2,tc2}	{2,4,4,0}	{ta2}	{ta2,2}
18	**17**	**{ta1,ta2,td2,tr2}**	**{4,0,6,2}**	**{td2}**	**{td2,0}**
19	**13**	**{ta1,ta2,td1,td2,tp1,tp2}**	**{0,2,0,2,0,0}**	**{tp2}**	**{tp2,0}**
20	14	{ta1,ta2,td1,td2,tc2}	{0,2,0,2,0}	{tc2}	{tc2,3}
21	15	{ta1,ta2,td1,tp1}	{3,5,3,0}	{tp1}	{tp1,0}
22	16	{ta1,ta2,td1,tc1}	{3,5,3,0}	{ta2}	{ta2,1}
23	17	{ta1,ta2,td1,td2,tc1,tr2}	{4,0,4,0,1,0}	{tc1}	{tc1,1}
24	18	{ta1,ta2,tr2,td2}	{5,1,1,1}	{tr2}	{tr2,1}
25	19	{ta1,ta2,tp2,td2}	{6,2,0,2}	{tp2}	{tp2,0}
26	20	{ta1,ta2,tc2,td2}	{6,2,0,0}	{ta1}	{ta1,2}
27	21	{ta2,tr1,td1,td2,tc2}	{4,0,0,4,2}	{tr1}	{ta1,0}
28	22	{ta2,tc2,td1,td2}	{4,2,0,4}	{tc2}	{tc2,1}
29	23	{ta2,td1,tp1}	{5,1,0}	{tp1}	{tp1,0}
30	24	{ta2,td1,tc1}	{5,1,0}	{ta2}	{ta2,1}
31	25	{td1,tc1,td2,tr2}	{2,1,0,0}	{tc1}	{tc1,1}
32	26	{td2,tr2}	{1,1}	{tr2}	{tr2,1}
33	27	{td2,tp2}	{2,0}	{tp2}	{tp2,0}
34	28	{td2,tc2}	{2,0}	{tc2}	{tc2,3}
35	**29**	**{tend}**	**{0}**	**{tend}**	**{tend,0}**

6.3. Pre-Runtime Scheduling Algorithm

The algorithm proposed in this work is a depth-first search method on a TLTS. So, the TLTS is not completely generated before the search, but it is partially generated, as needed. The TLTS is reduced since not all transitions are evaluated due to the partial-order reduction method and the undesirable transitions (which certainly lead to undesirable states). The *stop criterion* is obtained whenever the desirable final marking M^F is reached, representing that a *feasible firing schedule* was found.

The algorithm (Fig. 5) describes the proposed solution. Considering that, (i) the Petri net model is guaranteed to be bounded, and (ii) the timing constraints are by definition bounded and discrete, this implies that the TLTS is finite and thus the proposed algorithm always finishes.

The only way the algorithm returns TRUE is when it reaches a desired final marking (M^F), implying that a feasible schedule was found (line 3). The state space generation algorithm is modified (line 5) to incorporate the pruning from the partial-order reduction technique, and removing transitions (T^E) that lead to undesirable states. `PT` is a set of an ordered pairs $\langle t, \theta \rangle$ representing for each firable transition (pruned) all possible firing time in the firing domain. The *tagging scheme* (lines 4 and 9) ensures that no state is visited more than once. The function `fire` (line 8) returns a new generated state (S') due to the firing of transition t at time θ. The algorithm

```
1  scheduling-synthesis(S, M$^F$,PN)
2  {
3    if (S.M = M$^F$) return TRUE;
4    tag(S);
5    PT = pruning(firable($S$));
6    if (|PT| = 0) return FALSE;
7    for each ($\langle t, \theta \rangle$ $\in$ PT) {
8      S'= fire(S, t, $\theta$);
9      if (untagged(S') $\wedge$ scheduling-synthesis(S',M$^F$,PN)){
10       add-in-trans-system (S,S',t,$\theta$);
11       return TRUE;
12     }
13   }
14   return FALSE;
15 }
```

Figure 5: Scheduling Synthesis Algorithm

generates a transition system when successfully returning from the recursive call. The feasible schedule is represented by a timed labeled transition system that is generated by the function `add-in-trans-system` (line 10). In case the system does not have a feasible schedule, the proposed method searches the whole reduced state space.

The actual implementation of the pre-runtime scheduler can be reduced to an executive cyclic using an execution table (or lookup table) that states, for each task, its respective starting time. This information came from the timed labeled transition systems generated by the proposed pre-runtime scheduling synthesis framework.

6.4. Application of the Algorithm

Table 1 depicts the algorithm execution (Figure 5) applied to the time Petri net model of Figure 3. In this table, we see for each reachable state, the respective enabled transition set, the clock values, the firable transition set, and the chosen transition to be fired at a specific time instant. At line 14 (state 13), two transitions (tp_1 and tp_2) are firable. As it can be seen, the decision taken in a state may change the firable sequences. In this specific situation, the possible execution of task $T1$ on the processor (choosing tp_1 for firing) is a wrong choice because, after that, task $T2$ misses its deadline (line 18). The algorithm *backtracks* to state 13 (line 19) and try another alternative, now granting the processor to the task $T2$ (now choosing tp_2 for firing). This new decision leads to a feasible schedule, since in the line 35 the firing of transition t_{end} reaches the desired final marking.

Table 2: Experimental results summary

Example	instances	state-min	found	time (s)	method
Simple Control Application	28	50	50	0.0002	DS
Robotic Arm	37	150	150	0.01	NP
Xu (example 3)	4	171	1566	0.78	P
Xu (figure 9)	5	281	2387	2.6	P
Mine Drainage Control	782	3130	3255	9.3	NP
Unmanned Ground Vehicle	**433**	**4701**	**14761**	**324**	**P**

7. Case Study: Unmanned Ground Vehicle

The proposed method for scheduling synthesis was applied in applications such as simple control application (that runs on a 4-processors), robotic arm control, mine drainage control, and two examples from Xu and Parnas [Xu and Parnas, 1990] showing that priority-based scheduling could not find a feasible schedule even if such schedule exists. Table 2 shows a summary of the experimental results. In that table, *instances* represent the number of tasks' instances. *state-min* is the minimum number of states to be verified, *found* counts the number of states actually verified for finding a feasible scheduling, *time* expresses the algorithm execution time in seconds, and *method* is the scheduling method chosen (preemptive, non-preemptive, or defined subtasks). The results presented were obtained for finding the first feasible schedule.

All experiments were performed on a dual Pentium-III 600 Mhz processors with 768 MB RAM, OS Linux, and compiler GCC 2.95.4. However, in order to depict the practical usability of the proposed scheduling in more details, it is used an unmanned ground vehicle (UGV) case study, based on the work of [Sieh et al., 2001]. Although this application was intended to show how to deal with transient fault tolerance, it is used here to demonstrate the feasibility of the proposed methodology.

In this case study, the type of vehicle is designed to traverse hazardous ground for collecting various kinds of data (data, images,...). A semi-autonomous capability for making local path decisions is triggered when it encounters unforeseen hazards, e.g., debris, rubble, land miles, etc. A UGV have to provide two main services: its own mobility, and collecting information of interest for the controllers. The first one includes functions such as steering, braking, and speed control as well as the aiding in planning local autonomous movement. The second service includes the capture of data sensors, such as infrared, uwave, radar, and so on.

Table 3: UGV Specification Model

Task	c_p	d_p	p_p	arrival	d_s	min_s	Inst.
Vehicle Braking	2	33	28	sporadic	60	250	100
Hazard Response - Local Path Planning	20	26	175	sporadic	200	250	16
Sensor Data Fusion	10	80	400	periodic			7
Steering Control Loop	4	40	40	periodic			70
Steering Set Point	2	5	56	sporadic	60	200	50
Velocity Control Loop	4	30	40	periodic			70
Velocity Set Point	2	5	56	sporadic	60	200	50
System Management	5	60	100	periodic			28
CPU Status	5	100	200	periodic			14
Electrical System Status	5	100	200	periodic			14
Power Train Status	5	100	200	periodic			14

Table 3 shows the timing constraints for each task in this case study. In accordance with [Sieh et al., 2001], the tasks are all independent ones. This task model has a processor utilization factor of about 61%, which is not very low. As it can be noted, the sporadic tasks have to be translated into periodic ones. This translation is conducted in such a way that the LCM is minimized. The last column presents the number of task instances for each task, taking into accout that the LCM is equal to 2,800. In this case, the total of tasks' instances is 433. The intermediate model is generated using a specific tool that automatically translates the initial specification into a time Petri net model specified using the PNML (Petri Net Markup Language) formal [Weber and Kindler, 2003].

The proposed approach finds a feasible schedule after analyzing 14761 states, where the minimum number of states is 4701, in 324 seconds. It is worth noting the scheduling method is *all-preemptive*, implying that all tasks are implicitly split into all possible *subtasks*, which certainly increases time and space complexity. However, the time performance is not a major concern mainly due to the fact that the algorithm compute the schedule at compile time. Another reason, is due to the application domain considered (i.e. embedded systems), where schedule modifications are not common. The prototype tool that implements the algorithm should be optimized, mainly with respect to the tagging scheme.

8. Conclusions

This paper proposed a formal modeling methodology based on time Petri nets, and a framework for pre-runtime scheduling synthesis using a reduced state space exploration algorithm. The practical usability was depicted by a real-world case study, namely, an unmanned ground vehicle.

In spite of this analysis technique is not new, to the best of our knowledge, there is no work reported similar to ours that models hard real-time systems and finds (whether one exists) a respective pre-runtime scheduling.

The real-time task specification can be very general, since it can have timing constraints, and arbitrary intertask relations, such as precedence and exclusion relations. The formal modeling methodology presented how to model real-time systems using the time Petri net formalism, and also how to deal with undesirable states, for example, states with missed deadline.

The proposed algorithm is a depth-first search method on a finite timed labeled transition system derived from a time Petri net model. When searching for a feasible schedule, the algorithm suffers from the state space explosion problem. In order to maintain the state space growth under control, the proposed method uses minimization techniques. Considering the boundedness of the proposed time Petri net model, the algorithm presented always finds a schedule, provided that one exists.

The system modeling and pre-runtime scheduling synthesis proposed can be used to synthesize predictable and timely scheduled code. So, it is planned to generate complete executable code from the formal model. This can be solved through time Petri nets with tasks, which is an extension of time Petri nets, that annotates transitions with program code. Another extension is to add some flexibility to the pre-runtime scheduling. This problem can be solved through providing a small runtime scheduler for selecting different operational modes [Fohler, 1994], where each operational mode has a different pre-runtime scheduler associated.

References

Abdelzaher, T. F. and Shin, K. G. (1997). Comments on a pre-run-time scheduling algorithm for hard real-time systems. *IEEE Trans. Soft. Engineering*, 23(9):599–600.

Abdelzaher, T. F. and Shin, K. G. (1999). Combined task and message scheduling in distributed real-time systems. *IEEE Trans. Parallel and Distributed Systems*, 10(11):1179–1191.

Altisen, K., Göbler, G., Pnueli, A., Sifakis, J., Tripakis, S., and Yovine, S. (1999). A framework for scheduler synthesis. *IEEE Real-Time System Symposium*, pages 154–163.

Barreto, R., Maciel, P., Neves, M., Tavares, E., and Lima, R. (2004). A novel approach for off-line multiprocessor scheduling in embedded hard real-time systems. In *IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES'04)*. IFIP World Computer Congress.

Fohler, G. (1994). *Flexibility in Statically Scheduled Hard Real-Time Systems*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. Vienna, Austria.

Godefroid, P. (1994). *Partial Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* PhD Thesis, University of Liege.

Merlin, P. and Faber, D. J. (1976). Recoverability of communication protocols: Implicatons of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036–1043.

Mok, A. K. (1983). *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.* PhD Thesis, Dept Electrical Engineering and Computer Science, MIT.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proc. IEEE*, 77(4):541–580.

Shepard, T. and Gagné, J. A. (1991). A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Trans. Soft. Engineering*, 17(7):669–677.

Sieh, L., Haniak, P., and Richardson, P. (2001). Implementing transient fault tolerance in embedded real-time systems. In *IEEE Electronics and Information Technology Conference.*

Valmari, A. (1998). The state explosion problem. *LNCS: Lectures on Petri Nets I: Basic Models*, 1491:429–528.

Weber, M. and Kindler, E. (2003). The petri net markup language. *LNCS. Advances in Petri Nets. Petri Net Technology for Communication Based Systems*, 2472.

Xu, D., He, X., and Deng, Y. (2002). Compositional schedulability analysis of real-time systems using time petri nets. *IEEE Trans. Soft. Engineering*, 28(10):984–996.

Xu, J. and Parnas, D. (1990). Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. Soft. Engineering*, 16(3):360–369.

Xu, J. and Parnas, D. (1993). On satisfying timing constraints in hard real-time systems. *IEEE Trans. Soft. Engineering*, 1(19):70–84.