Quasi-Static Scheduling of Embedded Software Using Equal Conflict Nets

Marco Sgroi[1], Luciano Lavagno[2], Yosinori Watanabe[3], Alberto Sangiovanni-Vincentelli[1]

[1] University of California at Berkeley, EECS Department, Berkeley, CA 94720
[2] Cadence Berkeley Labs, 2001 Addison Street, Berkeley, CA 94704-1144
[3] Cadence European Labs, via S.Pantaleo 66, 00186 Roma, Italy

Abstract

Embedded system design requires the use of efficient scheduling policies to execute on shared resources, e.g. the processor, algorithms that consist of a set of concurrent tasks with complex mutual dependencies. Scheduling techniques are called *static* when the schedule is computed at compile time, *dynamic* when some or all decisions are made at run-time. The choice of the scheduling policy mainly depends on the specification of the system to be designed. For specifications containing only data computation, it is possible to use a fully static scheduling technique, while for specifications containing data-dependent control structures, like the if-then-else or while-do constructs, the dynamic behaviour of the system cannot be completely predicted at compile time and some scheduling decisions are to be made at run-time. For such applications we propose a Quasi-static scheduling (QSS) algorithm that generates a schedule in which run-time decisions are made only for data-dependent control structures. We use Equal Conflict (EC) nets as underlying model, and define quasi-static schedulability for EC nets. We solve QSS by reducing it to a decomposition of the net into conflict-free components. The proposed algorithm is complete, in that it can solve QSS for any EC net that is quasi-statically schedulable.

1 Introduction

Embedded systems are informally defined as a collection of programmable units, e.g. microcontrollers and DSPs, surrounded by ASICs and other standard components, that interact with the environment through sensors and actuators. Software development and its integration with the hardware is one of the main sources of cost in embedded system design. Modern design flows allow the designer to start with a functional specification of the overall system and map it onto a heterogeneous architecture including processors, memories and ASICs. However, an implementation on a shared resource, such as a processor, requires one to solve a *scheduling* problem, in order to sequence the execution of the concurrent modules while simultaneously (1) satisfying real-time constraints and (2) using the processor and the memory resources as efficiently as possible.

Embedded systems specifications usually contain both data computations and control structures. Control structures can be of two types: (1) data-dependent controls, like if-then-else or while-do loops, determine the next operation to be executed by testing the *value* of some data, and (2) real-time controls, like preemption and suspension, trigger actions after the *occurrence* of external or internal events. For specifications containing only data computations (assignments and fixed iteration loops) the schedule can be completely computed at compile time, and therefore is called *static*. A static schedule, usually implemented as a single task, is predictable and can be executed with almost no run-time overhead. However, when specifications include also some control it is not possible to compute the entire schedule at compile time. If the specification contains only data-dependent type of control in addition to data computation, the order in which operations are executed depends on the value of some data, which is known at run-time. In this case, *quasi-static* scheduling [4] techniques compute most of the schedule at compile time, leaving at run-time only the solution of data-dependent choices. Furthermore, if the specification allows communication via queues of unbounded size (e.g., in SDL or Dataflow networks), (quasi)static scheduling can bound the maximum size of those queues and ensure correct execution on an embedded system with a finite amount of physical memory. For specifications containing also real-time controls, the run-time behaviour heavily depends on the occurrence of external events. In this case, classical Real-Time scheduling techniques can be used to decide at run-time which tasks should be executed in reaction to such events. This type of schedule is called *dynamic*.

In this paper we present a new approach to schedule specifications including not only data-processing computations but also data-dependent control structures. We propose a *Quasi-static scheduling* (QSS) algorithm that takes as input a Petri Net (PN) model of the specification and produces as output a software implementation consisting of a set of concurrent tasks. The scheduling technique is quasi-static in the sense that, although any control decision is still made at run-time based on the value of the data, fragments of code that need to be executed as the consequence of the resolution are scheduled at compile time. Our technique addresses the problem of partitioning the functionality of the specification into tasks, i.e. functional blocks having the same execution rate. In particular, the algorithm we propose finds automatically a partition with minimum number of tasks, which is a problem usually solved by hand by experienced designers, and therefore allows to significantly reduce the run-time overhead and improve performances in single processor architectures. Here we address the problem of identifying tasks and deriving the code for each task, while issues like real-time dynamic scheduling of tasks are out of the scope of this paper.

We have chosen PNs as underlying formal model, because they allow to express concurrency, non-deterministic choice, synchronization and causality and because most properties are decidable for PNs. We represent data computations using *transitions* and channels between computation units using *places*. Data-

[4] Static and quasi-static scheduling generates sequential code and statically allocates communication buffers. Hence it is often called “software synthesis” in the literature, as opposed to the term “scheduling”, that is often reserved to dynamic real-time scheduling.

dependent control is modeled by places, called *choices*, with multiple output transitions, one for each possible resolution of the control. Data are modeled as tokens passed by transitions through places. In particular, we use a sub-class of PNs called *Equal Conflict Nets* (EC nets), because they exhibit clear distinction between the notions of concurrency and choice. Hence, they are appropriate to model computations in which control decisions depend on the value rather than on the arrival time of a token [5].

In this paper we introduce the notion of schedulability for EC nets. Informally, an EC net is quasi-statically schedulable if for *every resolution* of the control at the choice places, there exists a finite *cyclic firing sequence* that returns the tokens of the net to their initial places. The existence of cyclic sequences is required because it ensures that the number of data tokens that accumulate in any place is bounded even for infinite execution. We present an algorithm that first checks schedulability of the net to verify the correctness of the specification. If the net is not schedulable, the designer is notified that there exists no implementation that can be executed forever with bounded memory. If the net is schedulable, the algorithm computes a quasi-static schedule by *decomposing* the net into conflict free components and then applies static scheduling techniques to each component.

Previous work has used the decomposition of Free-Choice (FC) PNs into conflict free components. The fundamental theorem of Hack [9] on live and safe strongly connected FCPNs is based on the decomposition of the net into as many Marked Graphs (MGs) reductions as the number of the possible allocations of the non-deterministic choices. In [3] Best proposes an iterative algorithm to decompose a strongly connected ordinary PN into a set of strongly connected MGs. More recently, Teruel in [6] extends to weighted nets known results for ordinary nets [1]. These works have their main application in checking whether a given strongly connected net is bounded. However, in the domain of embedded reactive systems applications usually have lots of interactions with the environment, that are naturally modeled as source and sink transitions. As a result, *nets modeling embedded systems are not strongly connected.* Moreover, boundedness is a too restrictive property for our objective, that is finding *one* implementation with guaranteed bounded memory execution. In fact, schedulability implies the *existence* of at least one valid schedule that ensures that there is no unbounded accumulation of tokens in any place, while boundedness implies that *for all* the reachable markings, the number of tokens in any place does not exceed a certain number k. Therefore, we modify Hack's MG decomposition algorithm and apply it to the class of EC nets that have source and sink transitions.

Finally, our technique derives a software implementation of the schedule by traversing it and replacing transitions with the corresponding code (inline coding). A different approach to software synthesis using PNs has been recently proposed [8]. The algorithm in [8] generates a software program from a concurrent process specification through an intermediate PN representation. This

[5] EC nets model data-dependent control by abstracting the if-then-else control decisions as non-deterministic choices.

approach is based on the strong assumption that the PN is safe, i.e. buffers can store at most one data unit. This on one side guarantees termination of the algorithm, on the other side it makes impossible to handle multirate specifications, like FFT computations and downsampling. Moreover, safeness excludes the possibility to use PNs where source and sink transitions model the interaction with the environment: this makes it impossible to specify inputs with independent rate [6]. This method, assuming a priori that the net is schedulable, does not address also the important problem of verifying the schedulability of the specification.

The paper is organized as follows. In Section 2 we recall some definitions of the PN model and in Section 3 we shortly describe known techniques for scheduling of Weighted T-Systems. Section 4 defines the Quasi-Static Scheduling problem for EC nets and presents an algorithm to find a solution, if there exists one. Then, we describe how to generate a C program in Section 5. Section 6 presents an ATM Server application and experimental results.

2 Petri Nets: background

A **Petri Net** is a triple (P, T, F), where P is a non-empty finite set of places, T a non-empty finite set of transitions and $F : (T \times P) \cup (P \times T) \rightarrow \mathbb{N}$ the weighted flow relation between transitions and places. A **Petri Net graph** is a representation of a Petri Net as a bipartite weighted directed graph. If $F(x, y) > 0$, there is an arc with weight $F(x, y)$ from node x to node y. Given a node x, either a place or a transition, its **preset** is defined as ${}^\bullet x = \{y|(y, x) \in F\}$ and its **postset** as $x^\bullet = \{y|(x, y) \in F\}$. For a node y, $Pre[X, y]$ is a vector whose i-th component is equal to $F(x_i, y)$. A transition (place) whose preset is empty is called **source transition** (place), a transition (place) whose **postset** is empty is called **sink transition** (place). A place p such that $|p^\bullet| > 1$ is called **choice** or **conflict**. If $|{}^\bullet p| > 1$, p is called **merge**. Two transitions t and t' of a net N are said to be in **Equal Conflict Relation** [7] if $Pre[P, t] = Pre[P, t'] \neq 0$. A **marking** μ is an n-vector $\mu = (\mu_1, \mu_2, ..., \mu_n)$ where $n = |P|$ and μ_i is the non-negative number of tokens in place p_i. A transition t such that each input place p_i is marked with at least $F(p_i, t)$ tokens is enabled and may fire. When transition t fires, $F(p_i, t)$ tokens are removed from each input place p_i and $F(t, p_j)$ tokens are produced in each output place p_j.

The following PN properties, that are decidable for any Petri Net [11], are relevant in our discussion: **1) Reachability**. A marking μ' is reachable from a marking μ if there exists a firing sequence σ starting at marking μ and finishing at μ'. **2) Boundedness**. A Petri Net is said to be k-bounded if the number of tokens in every place of a rechable marking does not exceed a finite number k. A

[6] Two inputs have independent rate if their rates are not rationally related. An example of inputs with independent rate are the input keys from a keyboard. Instead, two streams of PCM samples for stereo audio are inputs with dependent rate.

[7] The Equal Conflict Relation is an equivalence relation that partitions the set of transitions of the net into a set of equivalence classes called Equal Conflict Sets [6].

safe Petri Net is one that is 1-bounded. **3) Deadlock-freedom**. A Petri Net is deadlock-free if, no matter what marking has been reached, it is possible to fire at least one transition of the net. **4) Liveness**. A Petri Net is live if for every reachable marking and every transition t it is possible to reach a marking that enables t.

Definition 1. A PN is a **Weighted (W)T-System** if $\forall p \in P : |p^{\bullet}| = |p^{\bullet}| = 1$.

Definition 2. A PN is a **Conflict Free (CF) Net** if $\forall p \in P : |p^{\bullet}| \leq 1$.

Definition 3. A PN is an **Equal Conflict (EC) Net** if ${}^{\bullet}t \cap t^{\bullet} \neq 0 \Rightarrow Pre[P, t] = Pre[P, t']$.

3 Static Scheduling of Weighted T-Systems

Weighted T-Systems (WT-Systems) [8] is a subclass of PNs that can represent concurrency and synchronization but not conflict and, therefore, is used to model specifications including pure data computation. WT-Systems can be mapped to Synchronous Dataflow networks where transitions are actors and places arcs; hence, one can apply to WT-Systems the well-known techniques for SDF scheduling [2]. The approach proposed by Lee [2] to find a static schedule for an SDF graph is based on the notion of finite complete cycle.

Definition 4. Given a Petri Net and an initial marking, a **finite complete cycle** is a sequence of transition firings that returns the net to its initial marking.

Since both the number of transition firings in a finite complete cycle and the tokens produced by each firing are finite, the number of tokens that can accumulate in any place of the net during the execution is bounded. If such a finite complete cycle exists, the net can be executed forever with bounded memory by repeating infinitely many times this sequence of transition firings (figure 1). Therefore, in [2] a static schedule is a periodic sequence of transitions and the period is a finite complete cycle.

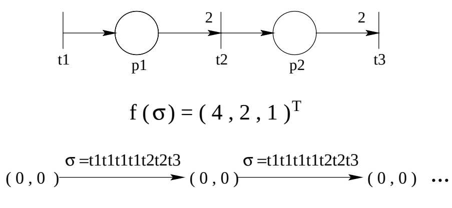

Fig. 1. Cyclic schedule

The problem of finding a finite complete cycle for a net can be reduced to the reachability problem, that is known to be decidable [11]. In terms of reachability

[8] Weighted T-Systems generalize Marked Graphs allowing arcs with non-unit weights

the goal is to find a sequence of transitions σ that starting from the marking μ returns the net to the same marking μ. To find σ, one must first solve the state equations [11] $f(\sigma)^T \cdot D = 0$. Here, D is a matrix where the i-th row corresponds to a transition t_i, the j-th column corresponds to a place p_j, and $D[i,j] = F(t_i, p_j) - F(p_j, t_i)$. A solution $f(\sigma)$, called **T-invariant** [1], is a vector whose i-th component $f_i(\sigma)$ is the number of times that transition t_i appears in sequence σ.

Definition 5. A Petri Net is **consistent** iff $\exists f(\sigma) > 0$ s.t. $f(\sigma)^T \cdot D = 0$.

The existence of a T-invariant is a necessary, but not sufficient condition for a finite complete cycle to exist. In fact, even if there exists a solution of the state equations, deadlock can still occur if there are not enough tokens to fire any transition. Therefore, once a T-invariant $f(\sigma)$ is obtained, it is necessary to verify by simulation that there exists a sequence σ_i that contains transition t_j as many times as $f_j(\sigma)$ and such that the net does not deadlock during execution. Such a sequence σ_i, if it exists, is a finite complete cycle. Lee [2] has shown that it is sufficient to simulate the firing sequences corresponding to the minimal vector [9] in the one-dimensional T-invariant space. This approach can be adopted for WT-Systems, but it is not adequate for larger classes of PNs containing non-deterministic choices [10].

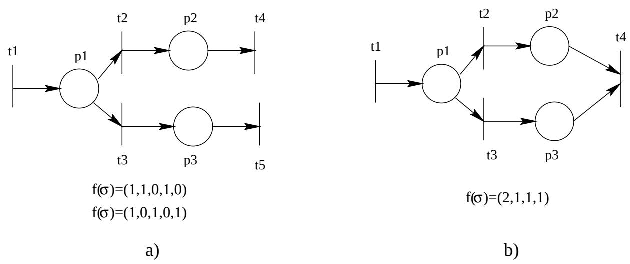

Fig. 2. Schedulable (a) and not schedulable (b) EC nets

4 Quasi-Static Scheduling of ECN

4.1 Definition of schedulability

Let $\Sigma = \{\sigma_1, \sigma_2 ...\}$ be a non-empty finite set of finite firing sequences such that for all $\sigma_i \in \Sigma$, σ_i is a finite complete cycle. Let σ_i^j be the j-th transition in

[9] A T-invariant is **minimal** when its set of non-zero entries is not a strict superset of that of any other T-invariant and the greatest common divisor of its elements is one [6].

[10] Nets containing non-deterministic choices may have two or more T-invariants that are linearly-independent (figure 2a).

sequence $\sigma_i = (\sigma_i^1 \sigma_i^2 ... \sigma_i^{j-1} \sigma_i^j \sigma_i^{j+1} ... \sigma_i^{N_i})$ and let Θ be the characteristic function of the Equal Conflict Relation, i.e. $\Theta(t, t') = 1$ iff t and t' are in Equal Conflict Relation.

Definition 6. The set Σ is a **valid set of finite complete cycles** if:
(1) $\forall \sigma_i \in \Sigma, \forall \sigma_i^j \in \sigma_i$ s.t. $\sigma_i^j \neq \sigma_i^h \ \forall h < j, \forall t_k \in T$ s.t. $t_k \neq \sigma_i^j$ and $\Theta(t_k, \sigma_i^j) = 1$, $\exists \sigma_l \in \Sigma$ s.t.
  (a) $\sigma_l^m = \sigma_i^m, \ \forall m \leq j - 1$
  (b) $\sigma_l^m = t_k$, m=j
(2) $\forall \sigma_i \in \Sigma$, there exists no transition $t \notin \sigma_i$ that is always enabled when σ_i is executed (fairness).

This definition informally means that for each sequence σ_i that includes a conflict transition σ_i^j, for each transition t_k that is in Equal Conflict Relation with σ_i^j, there exists another sequence σ_l s.t. σ_i and σ_l are identical up to the (j-1)th transition and have respectively σ_i^j and t_k at the j-th position in the sequence. Condition 2 (fairness) guarantees that there is no transition that, although always enabled during a cycle, is never executed. This definition allows for some transitions to be dead; liveness checking is trivially done once T-invariants have been derived.

Definition 7. Given an EC net N and an initial marking μ_0, a valid set of finite complete cycles Σ is **executable** if the net does not deadlock when its execution is simulated.

Definition 8. A valid set of finite complete cycles Σ is a **valid schedule** if it is executable.

Definition 9. Given an EC net N [11] and an initial marking μ_0, the pair (N,μ_0) is **(quasi-statically) schedulable**, if there exists a valid schedule.

This definition of schedulability extends to EC nets the concept of WT-System scheduling given in Section 3. If the net contains non-deterministic choices that model data dependent structures like if-then-else or while-do, a valid schedule is an executable set of firing sequences, one for every resolution of non-deterministic choices. A valid schedule must contain a finite complete cycle for every possible outcome of a choice because the value of the control tokens is unknown at compile time when the valid schedule is computed.

Executability is required because, in the case of strongly-connected EC net fragments, it is possible that successive choice resolutions belong to different branches, and hence accumulate tokens in various branches and cause deadlocks even when each subnet including only one branch by itself does not. An example is shown in Figure 3, where both the subnets including in one case $t_1 t_2 t_4 t_6 t_8$, in the other $t_1 t_3 t_5 t_7 t_8$, are fireable in isolation. However, when t_2 and t_3 fire in sequence, the complete net deadlocks (one token in p_2 and one token in p_3 are not sufficient to fire any transition).

[11] We consider only EC nets whose underlying undirected graph is connected, i.e. nets such that there exists an undirected path between every pair of nodes.

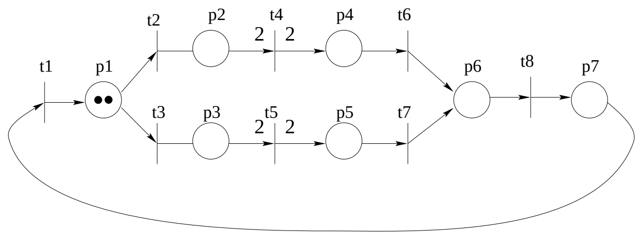

Fig. 3. EC net without executable Σ

We can also consider the scheduling problem as a game played against an adversary who can arbitrarily choose among conflicting transitions and has the goal of accumulating infinitely many tokens in any place of the net. Then, our objective is to find a non-terminating bounded memory execution by matching his choices with a cyclic schedule that returns the net to the initial marking.

Let us consider three examples. Given the net in figure 2a, $\Sigma = \{(t_1 t_2 t_4), (t_1 t_3 t_5)\}$ is a valid schedule because it contains a firing sequence for every value of the tokens in place p_1, i.e., whatever conflicting transition fires among t_2 and t_3, it is possible to complete the cycle that returns the net to the initial marking by firing t_4 or t_5 respectively. The net shown in figure 2b is not schedulable because there is no set of finite complete cycles that satisfies Definition 6. For example the set $\Sigma = \{(t_1 t_2 t_1 t_3 t_4), (t_1 t_3 t_1 t_2 t_4)\}$ is not valid because it does not contain any firing sequence beginning with $t_1 t_2 t_1 t_2$. This corresponds to the fact that, in case transition t_2 (t_3) is always fired, there exists no firing sequence that returns the net to the initial marking and therefore unbounded accumulation of tokens occurs in place p_2 (p_3).

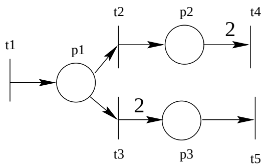

Fig. 4. Schedulable EC net with weighted arcs

The net shown in figure 4 is schedulable and $\Sigma = \{(t_1 t_2 t_1 t_2 t_4), (t_1 t_3 t_5 t_5)\}$ is a valid schedule. The weight two on the input arc of transition t_4 implies that t_2 has to fire twice before transition t_4 is enabled. However, there is no guarantee that this happens within a cycle because it is not possible to know a priori which transition among t_2 and t_3 fires. So, if transitions $t_1 t_2 t_1 t_3 t_5 t_5$ fire in this order, one token remains in place p_2 and the net does not return to the initial marking. The net is considered schedulable because repeated executions of this sequence do not result in unbounded accumulation of tokens (as soon as there are two tokens in place p_2, transition t_4 is fired and the tokens are consumed). The last example shows that a valid schedule does not necessarily include all the possible cyclic firing sequences, some even of infinite length, that can occur depending on the resolution of the non-deterministic choices (in this case the set would be $\{t_1t_3t_5t_5, t_1t_2(t_1t_3t_5t_5)^n t_1t_2t_4, \forall n \in \mathbb{N} \cup \{\infty\}\}$). A valid schedule should be intended only as a complete set of cyclic firing sequences that ensure bounded memory execution of the net. The set is complete in the sense that it is possible to derive from it a C-code implementation of the schedule including all the sequences that can occur, as we discuss in Section 5.

4.2 How to find a valid schedule

The algorithm is composed of two steps: 1) find a valid set of finite complete cycles, 2) check if the set is executable. Let us consider step 1. To find a valid set of finite complete cycles the net is first decomposed into as many Conflict Free (CF) components as the number of possible solutions for the non-deterministic choices of the net. Then, each component is statically scheduled. If every component is schedulable, we take a set that contains one finite complete cycle for each CF component. If at least one of the CF components is not schedulable, the net itself is said to be not schedulable.

Definition 10. Let $N = (T, P, F)$ and $N' = (T', P', F')$ be two PNs. N' is a **subnet** of N if $T' \subseteq T$, $P' \subseteq P$ and $F' = F \cap ((T' \times P') \cup (P' \times T'))$.

Definition 11. A subnet N' is a **transition-generated subnet** of N if P' is the set of all predecessors and successors places in N of the transitions in T' (i.e. $t \in T' \Rightarrow t^\bullet \subseteq P', {}^\bullet t \subseteq P'$).

Definition 12. A **T-component** N' of a net N is a set of transition-generated subnets such that each of them is a consistent Conflict Free net and for each initially enabled transition t in N', there exists a T-invariant containing t.

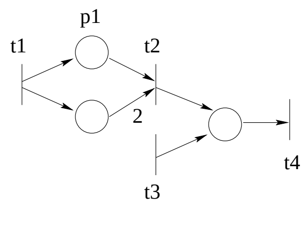

Fig. 5. Conflict Free net

The net in figure 5 is not a T-component: it is consistent because there exists a T-invariant ($f(\sigma) = (0, 0, 1, 1)$), but there is no T-invariant containing transition t_1. This corresponds to unbounded accumulation of tokens produced by the source transition t_1 occurring in place p_1.

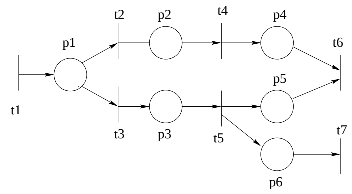

Fig. 6. Not schedulable EC net

Definition 13. A T-component N' is (statically) **schedulable** if there exists a firing sequence that returns it to the initial marking without any deadlock when its execution is simulated.

Definition 14. A **T-allocation** over an EC net is a function $\alpha : P \rightarrow T$ that chooses exactly one among the successors of each place $(\forall p \in P, \alpha(p) \in p^\bullet)$.

Definition 15. The **T-reduction** associated with a T-allocation is a set of subnets generated from the image of the T-allocation using the following Reduction Algorithm.

Let C_1,C_2, ...C_M be the Equal Conflict Sets defined in Section 2.3 as the equivalence classes of the Equal Conflict Relation and α_i the i-th T-allocation containing at most one transition for every C_j. The T-reduction $R_i = (T_{R_i}, P_{R_i}, F_{R_i})$ corresponding to T-allocation α_i is generated as follows (see figure 7).
Reduction Algorithm (modified from [9])

1. $R_i = N$ $(T_{R_i} = T,\ P_{R_i} = P,\ F_{R_i} = F)$.
2. For all $t_k \in T_{R_i}$ and $t_k \notin \alpha_i$
   (a) Remove t_k.
   (b) $\forall s \in t_k^\bullet$, remove place s unless one of the following conditions holds:
      i. s has a predecessor transition different from t_k $(\exists t \in^\bullet s$ s.t. $t \in T_{R_i})$.
      ii. the successor transition of s has a predecessor place that is different from s and is not a source place $(\exists t \in^\bullet (^\bullet(s^\bullet))$ s.t. $t \in T_{R_i})$.
   (c) If s_i is a removed place, $\forall t_j \in s_i^\bullet$, remove t_j if one of the following conditions holds:
      i. t_j has no predecessor place $(|^\bullet t_j| = 0)$.
      ii. all predecessors of t_j are source places. In this case remove every $s \in^\bullet t_j$.
   (d) Apply the previous two steps until they cannot be applied any longer.

Intuitively, the algorithm removes the part of the net that is inactive when the conflicting transitions included in the T-allocation are always chosen. Inactivity should be interpreted for a transition as the capability of firing only a finite number of times [9]. For places inactivity means having empty presets in the current net. Let us consider the EC net in figure 8 and apply the Reduction algorithm to compute the T-reduction from the T-allocation A_1 in figure 9.

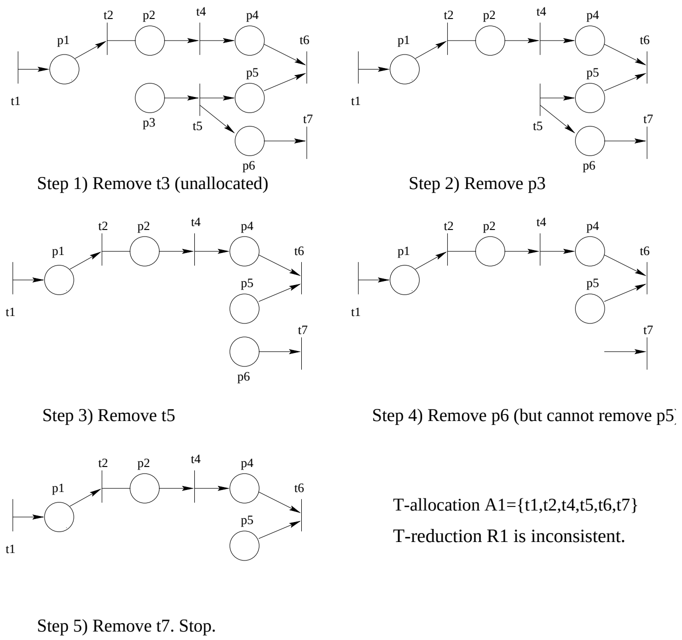

Fig. 7. How to obtain a T-reduction from the net shown in figure 6

First, unallocated transitions (t_3 and t_6) are removed. Then, successor places of every removed transition (in this case p_3) are deleted unless they are merge places with undeleted predecessor transitions or their successor is a transition feeded by at least one active places. The next step consists of the elimination of the transitions that are successors of places deleted in the previous step. At this point, transitions are deleted only if they have no predecessor places (so we can remove transition t_{10}) or all the predecessor places are inactive. The last two steps are iterated until no new node is deleted. In this example, the algorithm does not continue because place p_8 cannot be removed since it has a predecessor transition (t_{11}). When the final T-reduction obtained by applying this algorithm contains one or more inactive places, the net is not schedulable, since these places contain only a finite number of tokens and do not allow infinite execution.

Theorem 16. *The T-reduction R_i obtained by applying the reduction algorithm is*

1. *a set of transition-generated Conflict Free nets $\{R_i^1, R_i^2, ...R_i^M\}$.*
2. *a T-component of N iff every subnet R_i^l is consistent and for each initially enabled transition $t \in R_i$ there exists a T-invariant containing t.*

Proof. 1. Each R_i is a Conflict Free net by construction because it contains at most one transition for every Equal Conflict Set. Let's show that each subnet obtained from the Reduction Algorithm is transition-generated. We need to prove

that $\forall t_k \in R_{T_i}$ and $\forall s \in {}^\bullet t_k \cup t_k^\bullet$, $s \in R_{P_i}$. Consider places $s \in t_k^\bullet$, the algorithm does not remove a place s if $\exists t \in {}^\bullet s$ s.t. $t \in R_{T_i}$ (Reduction Algorithm (b)i). For places $s \in^\bullet t_k$, s is removed only if all other predecessor places of t_k are source places (Reduction Algorithm (b)ii). In this case at the next iteration step also t_k is removed (Reduction Algorithm (c)ii). Therefore, if $t \in R_{T_i}$ $t^\bullet \cup^\bullet t \subseteq R_{P_i}$.
2. Both directions trivially follow from part 1 of Theorem 16 and Definition 12.

Definition 17. An EC net N is **T-allocatable** if every T-reduction generated from a T-allocation is a T-component.

Theorem 18. *A set of finite complete cycles Σ that does not contain at least one finite complete cycle for every T-reduction is not valid.*

Proof. By contradiction. Assume that a set of finite complete cycles Σ not including any finite complete cycle for a T-reduction and including one finite complete cycle for any other T-reduction is valid. Consider two T-reductions: R_j derived from T-allocation A_j and R_i derived from A_i. R_i and R_j contain the same set of allocated transitions except for transitions t_m and t_n, such that $\Theta(t_m, t_n) = 1$ and $t_m \in R_i$ and $t_n \in R_j$. Since Σ is a valid set and includes one finite complete cycle σ_j for R_j, by Definition 6 it includes a sequence σ_x that is equal to σ_j upto transition t_n and has transition t_m instead of t_n, while all the other allocated transitions that appear in σ_j appear also in σ_x. Sequence σ_x can be a finite complete cycle only for R_i since no other T-reduction contains the same set of allocated transitions as R_j (except t_m). This contradicts the hypothesis that Σ contains no finite complete cycle for T-reduction R_i. Therefore, the assumption that Σ is valid was not true.

The following fundamental theorem states that T-allocatability, together with schedulability of every T-component, is a necessary and sufficient condition for the existence of a valid set of finite complete cycles. A net is T-allocatable if for all its components, each of them corresponding to a sequence of choices, there exists a cyclic schedule containing at least one occurrence of every initially enabled transition of the component. This means that a T-allocatable net, if there is no deadlock during execution of the cyclic schedules, can be executed forever with bounded memory, because for every choice there is always the possibility to complete successfully a finite cycle of firings that returns the net to the initial marking. Intuitively, T-allocations can be interpreted as control functions that choose which transition fires among several conflicting ones and therefore which component of the net is active at every cycle.

Theorem 19. *Given a EC net, there exists a valid set of finite complete cycles iff it is T-allocatable and every T-component is schedulable.*

Proof. ($\Rightarrow$) Let us prove that if the net is not T-allocatable or there exists a T-component that is not schedulable, there exists no valid set of finite complete cycles.

- Case 1: the net is not T-allocatable. This means that there exists at least one T-allocation A_i for which the generated T-reduction R_i is not a T-component. This may occur either if R_i is not consistent or if there exists at least an initially enabled transition $t_k \in R_i$ such that no T-invariant of R_i contains t_k. In the first case, i.e. R_i is not consistent, there is no finite complete cycle for the T-reduction R_i (i.e. for some non-deterministic choice resolutions there is unbounded accumulation of tokens). In the second case, i.e. there exists a finite complete cycle fcc_i for R_i but not including transition t_k, this finite complete cycle cannot be included in the valid set Σ because it does not satisfy the fairness condition of Definition 6. In both cases, any set Σ of finite complete cycles does not contain a finite complete cycle for T-Reduction R_i. For Theorem 18, Σ is not a valid set of finite complete cycles.
- Case 2. If a T-component R_i is not schedulable, there is no finite complete cycle for R_i and using the same argument above there exists no valid set of finite complete cycles.

($\Leftarrow$) By construction. The net is T-allocatable and each T-component is a schedulable Conflict Free net. We derive a valid set of finite complete cycles Σ, as follows: first of all, consider an arbitrarily chosen T-component TC_i and insert in Σ a finite complete cycle σ_i containing at least one occurrence of every transition in TC_i . Then, iterate for all sequences in Σ the following step: for every first occurrence of each conflicting transition $t_m \in \sigma_i$ insert in Σ a finite complete cycle σ_j for the T-component TC_j which contains the same allocated transitions as TC_i except for t_n instead of t_m. Choose σ_j so that it begins with the same subsequence as σ_i upto the different allocated transition where it contains t_n instead of t_m. Also, σ_j must contain at least one occurrence of every transition in TC_j. The set Σ derived following the above rules satisfies Definition 6. Condition (1) is trivially satisfied by construction. Condition (2) holds because σ_j contains at least one occurrence of every transition of TC_j and, since all the transitions of the net not in TC_j were removed in the Reduction process, they cannot be enabled for all markings reached by σ_j.

To check if a given net is T-allocatable, it is necessary to verify that every T-reduction obtained from the Reduction Algorithm is a T-component. For this purpose one must solve the state equations for every subnet of the T-reduction and check consistency; in the case any subnet of the T-reduction contains merge places, it is also necessary to check for this subnet that every transition initially enabled is in the support of at least one T-invariant. To detect if a T-component is schedulable the simulation based technique for WT-Systems described in Section 3 is used. At this point, finite complete cycles are derived from the T-invariants.

In term of complexity, the number of T-reductions (and therefore the number of applications of the Reduction Algorithm that computes them) is exponential in the number of conflicting transitions. However, simple heuristics can be used to reduce the cost of the algorithm in most practical cases [10]. The cost of computing a static schedule for each T-reduction is polynomial [2].

Let us describe two examples.

The EC net presented in figure 6 is not T-allocatable. Figure 7 shows the steps of the Reduction Algorithm applied to T-allocation $A_1 = \{t_1, t_2, t_4, t_5, t_6, t_7\}$. The generated T-reduction R_1, that is inconsistent because it contains an inactive place, is not a T-component and the net is not T-allocatable. Therefore, there exists no valid schedule for this EC net. In fact, if sequence $\sigma = (t_1 t_2 t_4)$ is fired infinitely often, there is unbounded accumulation of tokens in place p_4.

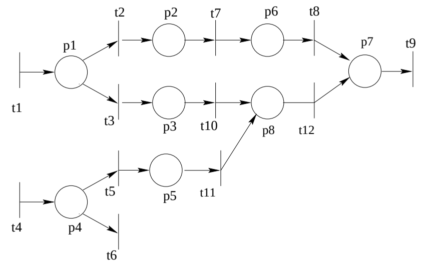

Fig. 8. Schedulable EC net

The EC net shown in figure 8 is T-allocatable. The T-components corresponding to the four T-allocations are represented in figure 9. To find a valid set of finite complete cycles we solve the state equations for each T-reduction. Let us consider T-Reduction R_1. The T-invariants of R_1 are $(1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0)$ and $(0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1)$. From the T-invariants a finite complete cycle is derived only after simulation checks that there is no deadlock. The same procedure is repeated for each T-reduction and a valid set of finite complete cycles for this EC net is $\{(t_1 t_2 t_7 t_8 t_9 t_4 t_5 t_{11} t_{12} t_9), (t_1 t_3 t_{10} t_{12} t_9 t_4 t_5 t_{11} t_{12} t_9), (t_1 t_2 t_7 t_8 t_9 t_4 t_6), (t_1 t_3 t_{10} t_{12} t_9 t_4 t_6)\}$.

Once a valid set of finite complete cycles is obtained, we need to check executability, i.e. whether there are enough tokens at the initial marking to execute all the finite complete cycles without a deadlock. The existence of a valid set of finite complete cycles guarantees by definition that each finite complete cycle is executable in the subnet in which the finite complete cycle was found. However, if the EC net contains a strongly connected subnet with a choice (figure 3), we must conduct an executability check for each of such structures since executability of each component does not imply executability of the whole net.

Consider the net shown in Figure 3. Two T-invariants are obtained as $(t_2 t_2 t_4 t_6 t_6 t_8 t_8 t_1 t_1)$ and $(t_3 t_3 t_5 t_7 t_7 t_8 t_8 t_1 t_1)$, where each is defined for a subnet corresponding to a simple cycle of the figure. When the executability of each T-invariant is checked for its subnet, we conduct a symbolic simulation. We assign a variable for each of the places in the subnet that are choices in the original EC net and have initial tokens, e.g. a variable $p_1^{(1)}$ at the place p_1 in the subnet for the first T-invariant above (upper cycle in the figure). We then simulate the T-invariant, using these variables as the initial numbers of tokens at such places while we use

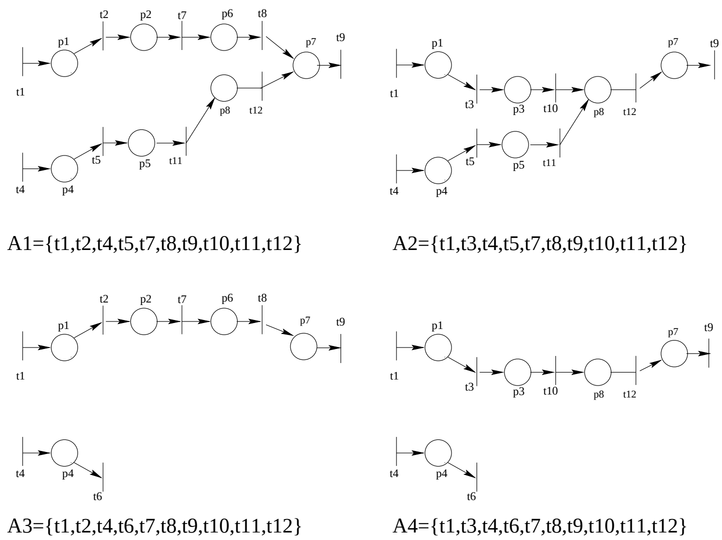

Fig. 9. T-components

the actual number of tokens for a place that is initially marked but no variable is defined for. The result of the simulation is an inequality of the variables such that given initial numbers of tokens allow the T-invariant to be executed if and only if the inequality holds. For our example, we obtain $p_1^{(1)} \geq 2$, which means that the place p_1 must have at least two tokens in order to execute the first T-invariant in its subnet. Similarly, for the second T-invariant, we obtain $p_1^{(2)} \geq 2$, where $p_1^{(2)}$ is the variable defined at p_1 for the corresponding subnet. Since p_1 is initially marked with two tokens, we also have an equation $p_1^{(1)} + p_1^{(2)} = 2$. Now, a value of each variable represents the number of tokens at the place that can be used to fire transitions in the corresponding subnet. Thus Σ is executable if for every integer assignment of these variables which satisfies the equation, at least one of the inequalities holds. Equivalently, Σ is *not* executable if there exists an integer solution which satisfies the equation but none of the inequalities. In this example, by reversing the directions of the inequalities, our problem is to find a solution for the system made of $p_1^{(1)} + p_1^{(2)} = 2$, $p_1^{(1)} < 2$, and $p_1^{(2)} < 2$. Since $p_1^{(1)} = p_1^{(2)} = 1$ is such a solution, we conclude that this valid set of finite complete cycles is not executable. In general, we obtain a set of inequalities and equalities, where each inequality is a boolean disjunction of linear equations and each equality is a linear equation. The problem can be solved by iteratively solving an integer programming. Alternatively, the equalities and inequalities may be represented using decision diagrams, e.g. [5], and the existence of a solution can be found with a constant time by tautology checking.

4.3 BDF and Petri Nets

The problem of scheduling specifications containing data-dependent control was first addressed in [7]. Buck and Lee [7] introduced the Boolean Data Flow (BDF) networks model and proposed an algorithm to compute a quasi-static schedule. However, the problem of scheduling BDFs with bounded memory is undecidable, i.e. any algorithm may fail to find a schedule even if the BDF is schedulable. Hence, the algorithm proposed by Buck can find a solution only in special cases. On the contrary, schedulability of PNs is decidable as a consequence of the decidability of the reachability problem [11]. PNs and Boolean Dataflow networks (BDF) [7], both used to model specifications containing data-dependent control, have different expressive power. One difference is in the semantics of the communication channels among blocks. Channels in BDFs are FIFO queues that preserve the order of the valued tokens. Instead, PNs do not have FIFO semantics because the tokens do not carry values [7]. This implies also that in the PNs model, differently from BDF, there can be no correlation among the results of different non-deterministic choices. The latter property in particular implies that some schedulable BDF nets are not schedulable when modeled as an EC net, just as some schedulable BDFs are not schedulable according to [7].

BDF is a determinate model, since all the valid executions of the network produce the same token streams regardless of the order in which the actors are executed [7], while PNs are not determinate when they contain merge places. However, it is possible to make a PN determinate by imposing appropriate conditions to the schedule. For example, the net shown in figure 10, is not determinate

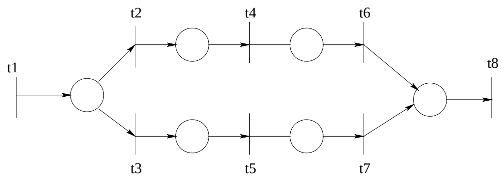

Fig. 10. Example of non-determinate PN

because of the presence of the non-deterministic merge, i.e. there is no guarantee on the order of the tokens in the output stream if t_6 and t_7 are concurrently enabled. To enforce determinacy, it is necessary to impose some restrictions to the scheduler and ensure that the states where two merging transitions are concurrently enabled are never reached. One way is to allow tokens to enter the net (i.e. firing t_1) only if no other transition is enabled. Further investigation in this direction is necessary to formally define and prove the conditions under which a PN is determinate.

5 C-code generation

The final goal of this paper is the synthesis of a software implementation [12] that satisfies functional correctness and minimizes a cost function of latency and memory size. In general, such implementation consists of a set of software tasks that are invoked by the Real Time Operating System either by interrupt or polling. Our approach for software synthesis derives an implementation directly from a valid schedule, that should be intended as an intermediate description, containing in explicit form a set of rules, such as number and order of firing of transitions, that an implementation should follow to guarantee bounded memory execution. In this Section we show how to generate from a valid schedule an implementation that consists of as many fragments of C code (*tasks*) as the number of source transitions with independent firing rate. Transitions with independent firing rate cannot be quasi-statically scheduled together and therefore a task is composed only of transitions with dependent firing rates, that are transitions belonging to the same T-invariant.

Schedule (Σ)
while $(t_i \neq EOS)$ { Task(Σ,i); $i = i + 1$; }

Task (Σ, i)
while $(t_i \neq EOT)$ { if t_i is already visited
  then{ insert **goto** label t_i }
  else{
    if t_i is a conflicting transition { insert **if..then..else** }
      if $f(t_i) < f(t_{i-1})$ { insert counting var and **if** test }
      if $f(t_i) > f(t_{i-1})$ { insert counting var and **while** test }
      if $f(t_i) = f(t_{i-1})$ { insert t_i } } }
end Task

Fig. 11. Pseudo-code of the code generation algorithm

A pseudocode description of the algorithm that generates C code is shown in figure 11 (EOS means End of Sequence and EOT means End of T-Invariant). The routine *Schedule* visits all the transitions in the valid schedule Σ, by calling the routine Task every time a new T-invariant is visited. *Task* checks if a transition has already been visited and, if so, inserts a label and a goto to avoid repetition of code. This corresponds to the presence of a merge place in the EC net model that yields code patterns which are common either to the branches of an *if-then-else* or are shared by different tasks. Instead, if the transition currently visited is a conflicting one, an if-then-else structure is generated and the code in the two branches is synthesized by traversing the two finite complete cycles of Σ containing the conflicting transitions. In case of multirate nets, a variable counting the

[12] We consider uni-processor architecures, that are used for most embeeded systems applications like cellular phones, multimedia set-top boxes etc.

number of tokens and a test are used to determine whether an operation should be executed. Figure 12 provides an example of C program generated for the net shown in figure 4 and whose valid schedule is $\Sigma = \{(t_1 t_2 t_1 t_2 t_4)(t_1 t_3 t_5 t_5)\}$.

```
while (true) {
  t1;
  if (p1) {
    t2; count(p2)++;
    if (count(p2) == 2) {
        t4; count(p2)- =2; }
      } else {
      t3; count(p3)+=2;
      while (count(p3 ≥ 1) {
        t5; count(p3)- -; } } }
```

Fig. 12. C code for the EC net of figure 4

6 Experimental Results

We applied our algorithm to synthesize a software implementation of a real life embedded system: an ATM server for Virtual Private Networks [4]. The main functionalities of the server are (1) a message discarding technique (MSD) that avoids node congestion and (2) a bandwidth control policy based on a Weighted Fair Queueing (WFQ) scheduling discipline. Figure 13a gives a high-level description of the algorithm. The inputs of the system are *Cell*, an interrupt that occurs at irregular times when a non-empty cell enters the Server and *Tick*, an event that periodically triggers the process of forwarding the next outgoing cell to the output port. Therefore, Cell and Tick are inputs with independent firing rate. The module MSD decides whether an incoming cell must be accepted and the module CELL_EXTRACT selects, every cell slot, which cell must be emitted among those stored in the internal buffer. WFQ_SCHEDULING may be activated either by MSD or by CELL_EXTRACT and computes the cell emission time. We have chosen this example because it implements a data-dominated algorithm containing several data-dependent control structures. We modeled the algorithm using a ECN containing 49 transitions and 41 places (figure 13b), of which 11 non-deterministic choices. From the ECN model we could derive a valid schedule containing 120 finite complete cycles, one for each different T-reduction, and from the valid schedule we obtained a software implementation composed of two tasks, one for each input with independent firing rate.

In table I we compare two software implementations: the first, named QSS, was obtained using our Quasi-Static Scheduling based technique and consists of two tasks, the second, named functional task partitioning, consists of five tasks and was obtained by synthesizing separately one tasks for each of the five modules shown in figure 13. The results, obtained using a testbench of 50 ATM

Sw implementation	QSS	Functional task partitioning
Number of tasks	2	5
Lines of C code	1664	2187
Clock cycles	197526	249726

Table I

cells, show that the number of clock cycles and the code size are significantly smaller for the QSS implementation that is composed of a smaller number of tasks and therefore has a smaller overhead due to tasks activation.

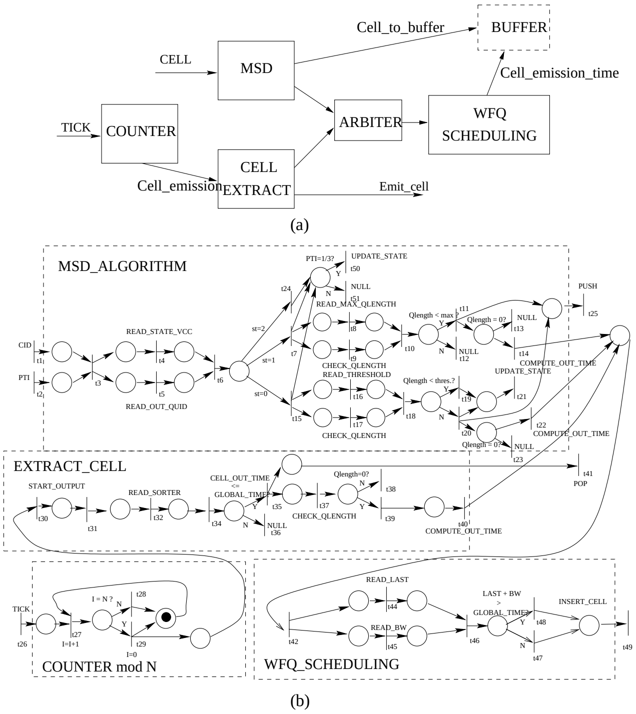

Fig. 13. ATM Server functional description (a) and ECN model (b)

7 Conclusions

In this paper we have defined schedulability for Equal Conflict Nets and presented a quasi-static scheduling (QSS) algorithm that finds a schedule, whenever there exists one. This result is important because QSS, maximizing the amount of work done at compile time, allows to reduce significantly the run-time scheduling overhead. Moreover, we have explained why EC nets are an appropriate model for embedded systems specifications containing data processing and data-dependent control, and compared them with other models. We have showed how a valid schedule can be found by extending well-established techniques used for scheduling Weighted T-Systems, and presented also how to generate a C code implementation of the schedule. The presented algorithms have been applied to a real case study, an ATM Server, and their potential has been demonstrated.

Acknowledgments

The authors want to thank Jordi Cortadella, Mike Kishinewsky, Alex Kondratyev, Howard Wong-Toi and Lothar Thiele for useful comments and discussions.

References

1. J. Desel and J.Esparza. *Free choice Petri nets.* Cambridge University Press, 1995.
2. E.A.Lee and D.G.Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Transactions on computers*, January 1987.
3. E.Best. Structure theory of petri nets: the free choice hiatus. In *Advances in Petri Nets*, 1986.
4. E.Filippi et al. Intellectual property re-use in embedded system co-design: an industrial case study. In *International Symposium System Synthesis. ISSS '98. Taiwan*, December 1998.
5. I.R.Bahar et. al. Algebraic decision diagrams and their applications. In *IEEE International Conference on Computer-Aided Design*, November 1993.
6. E.Teruel and M.Silva. Structure theory of equal conflict systems. In *Theoretical Computer Science, vol.153, pp. 271-300*, 1996.
7. J.Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model.* Ph.D dissertation. UC Berkeley, 1993.
8. B. Lin. Software synthesis of process-based concurrent programs. In *Proceedings of the Design Automation Conference*, June 1998.
9. M.Hack. *Analysis of Production Schemata by Petri Nets.* Master thesis. MIT, 1972.
10. M. Sgroi. *Quasi-Static Scheduling of Embedded Software Using Free-Choice Petri Nets.* M.S. dissertation. UC Berkeley, May 1998.
11. T.Murata. Petri nets: properties, analysis and applications. In *Proceedings of the IEEE*, April 1989.