Synthesis of Embedded Software Using Free-Choice Petri Nets

Marco Sgroi[†], Luciano Lavagno[‡],
Yosinori Watanabe[‡] and Alberto Sangiovanni-Vincentelli[†]

[†] University of California, Berkeley, CA
[‡] Cadence Design Systems

Abstract

Software synthesis from a concurrent functional specification is a key problem in the design of embedded systems. A concurrent specification is well-suited for medium-grained partitioning. However, in order to be implemented in software, concurrent tasks need to be scheduled on a shared resource (the processor). The choice of the scheduling policy mainly depends on the specification of the system. For pure dataflow specifications, it is possible to apply a fully static scheduling technique, while for algorithms 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 Free Choice Petri Nets (FCPNs), as underlying model, and define quasi-static schedulability for FCPNs. The proposed algorithm is complete, in that it can solve QSS for any FCPN that is quasi-statically schedulable. Finally, we show how to synthesize from a quasi-static schedule a C code implementation that consists of a set of concurrent tasks.

1 Introduction

Software development and its integration with the hardware is one of the main sources of cost in embedded systems design. For this reason, modern design flows allow the designer to start with a functional and implementation-independent specification of the overall system, and map it onto a heterogeneous architecture. Even in cases when software development is carried out directly in a high-level language such as C, it is often convenient to initially decompose the specification into concurrent modules. However, an implementation on a shared resource, such as a processor, requires one to solve a *scheduling* problem in order to sequence the operation of the concurrent modules while simultaneously (1) satisfying real-time constraints and (2) using the processor and 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 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 techniques compute most of the schedule at compile time, leaving at run-time only the solution of data-dependent decisions. Quasi-static scheduling should also partition the functionality of the specification into tasks, i.e. functional blocks having the same execution rate. Finding a partition with minimum number of tasks, which is a problem usually solved by hand by experienced designers, allows to significantly reduce the run-time overhead and improve performances in single processor implementations. 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*. For these reasons, an ideal scheduling technique should combine the best aspects of the scheduling techniques we described above. In particular it should use: (1) *static* scheduling to exploit fixed dependencies between blocks of operations, (2) *quasi-static* scheduling to identify data-dependent operations with the same rate and schedule them, (3) *dynamic* scheduling to determine which tasks, among those identified at the previous step, should be executed. Static and quasi-static scheduling generates sequential code and can statically allocate 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. An effective software synthesis technique should: (1) check if the specification can be scheduled in finite memory, and thus be implemented on an embedded processor, and (2) allow one to evaluate tradeoffs between memory size and execution speed of the final implementation.

Several techniques for software synthesis from a concurrent functional specification, along the lines discussed above, have been proposed. Buck and Lee [5] have introduced the Boolean Data

Flow (BDF) networks model and proposed an algorithm to compute a quasi-static schedule. However, the problem of scheduling BDF 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. Thoen et al. [3] proposed a technique to exploit static information in the specification and extract from a constraint graph description of the system statically schedulable clusters of threads. The limit of this approach is that it does not rely on a formal model and does not address the problem of checking whether a given specification is schedulable. Lin [6] proposed an algorithm that generates a software program from a concurrent process specification through an intermediate Petri-Nets representation. This approach is based on the strong assumption that the Petri Net 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. Safeness implies that the model is always schedulable and therefore also Lin's method does not address the problem of verifying schedulability of the specification. Moreover, safeness excludes the possibility to use Petri Nets where source and sink transitions model the interaction with the environment. This makes impossible to specify inputs with independent rate [1].

In this paper we propose a new approach to software synthesis. Our algorithm takes as input a Petri Nets (PN) model of the system and produces as output a software implementation consisting of a set of software tasks that are invoked at run-time by the Real-Time Operating System (RTOS). Here, we address the problem of identifying tasks and synthesizing the code for each of them, while RTOS issues like dynamic scheduling of tasks are out of the scope of this work. To identify tasks and synthesize the code for each task we consider only the part of the specification including data computation and data-dependent control and we compute a quasi-static schedule.

We have chosen PNs as underlying formal model, because they allow to express concurrency, non-deterministic choice, synchronization and causality and because most properties, including schedulability, are decidable for PNs. We represent data computations using a type of nodes (*transitions*) and non-FIFO channels between computation units using another type of nodes (*places*). Data-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 *Free-Choice* (FCPNs), because they exhibit clear distinction between the notions of concurrency and choice. Hence they are appropriate to model computations in which the outcome of a choice depends on the value rather than on the arrival time of a token.

In this paper we introduce the notion of schedulability for FCPNs. Informally, a FCPN is quasi-statically schedulable if for every possible resolution of the control at the choice places, there exists a cyclic finite 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 statically schedulable components. Then, it derives a software implementation by traversing the schedule and replacing transitions with the corresponding code.

[1]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.

The paper is organized as follows. In Section 2 we shortly describe the PN model and the notion of cyclic schedules. In Section 3 we define the QSS problem for FCPNs and present an algorithm to find a solution, if there exists one. Then, we describe how to generate a C program in Section 4. Section 5 presents an application (ATM Server) and experimental results.

2 Preliminaries

Petri Nets 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. [9] 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** if $Pre[P, t] = Pre[P, t'] \neq 0$ [4]. 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 properties, that are decidable for any Petri Net (PN), are relevant in our discussion [9]. **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'. **Boundedness**: a PN is said to be k-bounded if the number of tokens in every place of a reachable marking does not exceed a finite number k. A safe PN is one that is 1-bounded. **Deadlock-freedom**: a PN is deadlock-free if, no matter what marking has been reached, it is possible to fire at least one transition of the net. **Liveness**: a PN is live if for every reachable marking and every transition t it is possible to reach a marking that enables t.

The following are some of the most common subclasses of PNs. **Marked Graph**: a PN such that each place p has at most one input transition and one output transition. **Conflict Free Net**: a PN such that each place p has at most one output transition. **Free Choice Net**: a PN such that every arc from a place is either a unique outgoing arc or a unique incoming arc to a transition.

Marked Graphs can represent concurrency and synchronization but not conflict. Free Choice Nets allow one to model both conflict and synchronization, under the condition that every transition that is successor of a choice has exactly one predecessor place. This implies that whenever an output transition of a place is enabled, all the output transitions of that place are enabled (figure 1a)). Therefore, Free Choice Nets model data-dependent control by abstracting if-then-else control decisions as non-deterministic choices, but they can not model external conditions or multiple rendezvous. The PN shown in figure 1b) is not a Free Choice Net, because there exists a marking in which transition t_3 is enabled and transition t_2 is not.

Cyclic schedules Synchronous Dataflow (SDF) [1] networks are a special case of Petri Nets, since they can be mapped into Marked Graphs where actors are transitions and arcs places. The approach proposed by Lee [1] to find a static schedule for an SDF graph is based on the notion of finite complete cycle. Given a Marked Graph and an initial marking, or equivalently a SDF graph and an initial configuration of tokens, a **finite complete cycle** is a sequence of transition firings that returns the net to its initial marking. If such

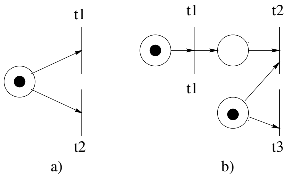

Figure 1: Free Choice Net(a), not Free Choice Net(b)

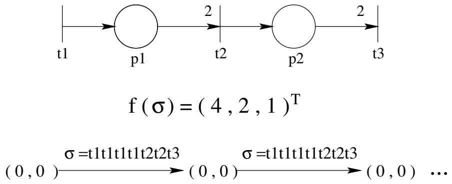

Figure 2: Cyclic schedule

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

To find a finite complete cycle σ, one must first solve the state equations [9] $f(\sigma)^T \cdot D = 0$. A solution $f(\sigma)$, called **T-invariant** [9], is a vector whose i-th component $f_i(\sigma)$ is the number of times that transition t_i appears in sequence σ.

Definition 2.1 *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: 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 [1] has shown that it is sufficient to simulate the firing sequences corresponding to the minimal vector in the one-dimensional T-invariant space. This approach is not adequate for larger classes of Petri Nets that contain non-deterministic choices [8]. The rest of this paper is devoted to the solution of the scheduling problem for Free Choice Nets.

3 Quasi-static Scheduling of FCPN

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 and contains at least one occurrence of each source transition of the net. Let σ_i^j be the j-th transition in 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 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 3.1 *The set* Σ *is a* **valid schedule** *if* $\forall \sigma_i \in \Sigma, \forall \sigma_i^j \in \sigma_i$ *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.*

(1) $\sigma_l^m = \sigma_i^m$, $\forall m \leq j-1$
(2) $\sigma_l^m = t_k$, *m=j*

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.

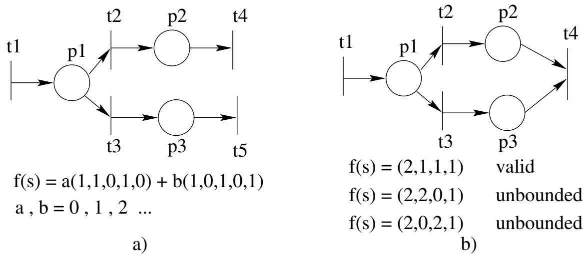

Figure 3: Schedulable (a) and not schedulable (b) FCPNs

Definition 3.2 *Given an FCPN N 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 non-static Data Flow networks the concept of SDF scheduling given in Section 2. If the net contains non-deterministic choices that model data dependent structures like if-then-else or while-do, a valid schedule is a set of finite complete cycles, 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.

Schedulability implies the **existence** of at least one valid schedule that ensures that there is no unbounded accumulation of tokens in any place. This property is different from k-boundedness, that implies that **for all** the reachable markings, the number of tokens in any place does not exceed a certain number k.

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.

Given the net in figure 3a, $\Sigma = \{(t_1 t_2 t_4), (t_1 t_3 t_5)\}$ is a valid schedule because for every solution of the conflict between transitions t_2 and t_3, it is possible to complete a cycle that returns the net to the initial marking by firing t_4 after t_2 or t_5 after t_3. Instead, the net shown in figure 3b is not schedulable because there exists no finite complete cycle if the conflict is always solved choosing t_2 (t_3). In fact, if the token values in p_1 are such that t_2 (t_3) is always fired, unbounded accumulation of tokens occurs in place p_2 (p_3).

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).

This 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 $\{t_1 t_3 t_5 t_5, t_1 t_2 (t_1 t_3 t_5 t_5)^n t_1 t_2 t_4$, $\forall n \in \mathbb{N} \cup \{\infty\}\}$). 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

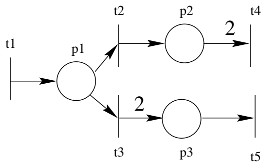

Figure 4: Schedulable Petri Net with weighted arcs

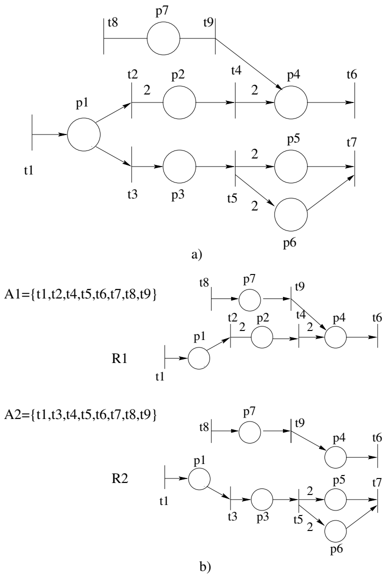

Figure 5: T-allocations and T-reductions

possible to derive from it a C-code implementation of the schedule including all the sequences that can occur, as we discuss in Section 4.

How to find a valid schedule 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. Then, each component is statically scheduled. If every component is schedulable, a valid schedule is a set of finite complete cycles, one for each CF component. If at least one of the CF components is not schedulable, the net itself is said to be not schedulable.

A similar approach, called MG decomposition, has been used by Hack [7] to check liveness and safeness of strongly connected ordinary FCPNs. More recently, Teruel [4] has extended to weighted nets a theorem that is used to check whether a given strongly connected net is bounded. However, in the domain of embedded reactive systems most 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, as we outlined in the previous section, boundedness is a too restrictive property for our objective. For this reason we modify Hack's MG decomposition algorithm and apply it to the class of FCPNs that have source and sink transitions.

Step 1. Decompose the net into Conflict Free (CF) components

Definition 3.3 *A* **T-allocation** *over a FCPN N 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 3.4 *The* **T-reduction** *associated with a T-allocation is a set of subnets generated from the image of the T-allocation using the Reduction Algorithm.*

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 (executed) at every cycle. If we consider the net shown in figure 5a), there exist two T-allocations, A_1 containing t_2 and A_2 containing t_3. During a cycle where t_2 is fired, only transitions t_1, t_4 and t_6 can be fired, while the rest of the net is not executed (figure 5b)). A T-reduction is the subnet obtained from N by removing the part that is inactive when the conflicting transitions included in the corresponding T-allocation are chosen. *A T-reduction is by construction a Conflict Free net.* More precisely, a T-reduction is a set of disjoint CF subnets, as shown in figure 5. The first step of the algorithm consists of computing all the T-reductions of the net. Given a net N and a T-allocation α_i, the corresponding T-reduction $R_i = (T_{R_i}, P_{R_i}, F_{R_i})$ is derived by applying the following Reduction Algorithm (modified from [7]; for further details see [8]).

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.*

Step 2. Check if every CF component is statically schedulable

Definition 3.5 *A T-reduction R_i is* **schedulable** *if (1)it is consistent, (2)for each source transition t_s in N, it has a T-invariant containing t_s, (3)there exists a firing sequence that returns R_i to the initial marking without any deadlock when its execution is simulated (generalization of [1]).*

A T-reduction R_i is a schedulable CF component if it has a finite complete cycle that contains at least one occurrence of every source transition of the net and returns R_i to the initial marking. Conflict Free nets do not contain any non-deterministic choice and therefore, to check if there exists a static schedule, it is possible to apply the standard techniques for SDF described in Section 2.

Step 3. Derive a valid schedule, if there exists one

The following theorem states that schedulability of each T-reduction is a necessary and sufficient condition for the existence of a valid set of finite complete cycles. A net is schedulable if for all its T-reductions, each of them corresponding to a sequence of choices, there exist a firing sequence containing at least one occurrence of every transition of the reduction. This means that a schedulable net, if there is no deadlock during execution of the cyclic schedules, can be executed forever with bounded memory, because for every resolution of the choices there is always the possibility to complete

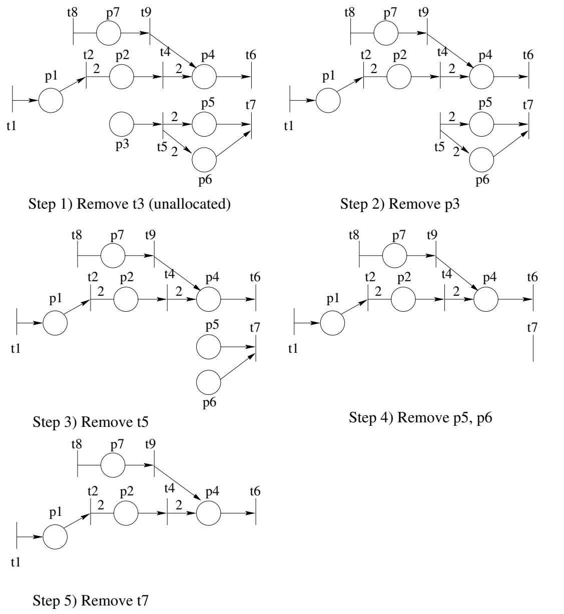

Figure 6: How to obtain T-reduction R_1 from the net shown in figure5

successfully a finite cycle of firings that returns the net to the initial marking [2].

Theorem 3.1 *Given a FCPN, there exists a valid schedule iff every T-reduction is schedulable.*

The proof can be found in [8].

The FCPN shown in figure 7 is not schedulable. Both T-reductions R_1 and R_2 are inconsistent because they contain a source place that corresponds to finite execution; in R_1 for example, if sequence $\sigma = (t_1 t_2 t_4 t_6)$ is fired infinitely often, there is unbounded accumulation of tokens in place p_4 since p_3 cannot provide infinitely many tokens. Therefore, the net is not schedulable.

The FCPN shown in figure 5 is schedulable. To find a valid set of finite complete cycles we solve the state equations for each T-reduction: the T-invariants of R_1 are $(1,1,0,2,0,4,0,0,0)$ and $(0,0,0,0,0,1,0,1,1)$. A finite complete cycle is derived from the T-invariants only after simulation checks that there is no deadlock in the corresponding T-reduction. The same procedure is repeated for each T-reduction and a valid set of finite complete cycles for this PN is $\{(t_1 t_2 t_4 t_4 t_6 t_6 t_6 t_6 t_8 t_9 t_6), (t_1 t_3 t_5 t_7 t_7 t_8 t_9 t_6)\}$.

In terms of complexity, the number of T-reduction is exponential in the number of conflicting transitions, while the cost of statically scheduling each T-reduction is polynomial [1]. The C code generation algorithm that we present in the next Section generates code that is linear in the size of the PN.

4 C-code generation

The ultimate goal is the synthesis of a software implementation 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 enabled by the occurrence of external events and are invoked by the Real Time Operating System either by interrupt or polling. Our software synthesis technique 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 any implementation should follow to guarantee bounded memory execution.

[2] If the net presents certain strongly connected PN fragments, it is possible that tokens accumulate in various T-invariants causing the net to deadlock even when each T-invariant by itself does not. In this case it is necessary to check the executability of the net using a technique described in [8]

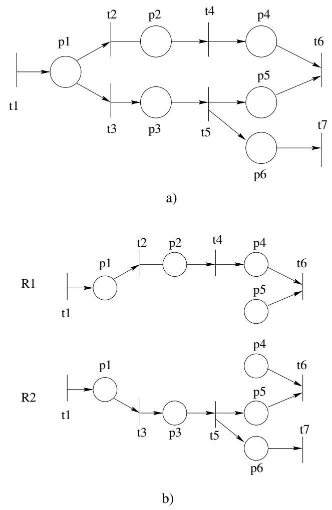

Figure 7: Non-schedulable FCPN

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. Generating one task for every input with independent firing rate identifies a lower bound in the number of tasks, because transitions with independent firing rate cannot be quasi-statically scheduled together and therefore cannot be included in the same task. A task is composed only of transitions with dependent firing rates, i.e. transitions belonging to the same T-invariant. The algorithm is as follows (EOS means End of Sequence and EOT means End of 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) { 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 } } }

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 PN 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 number of tokens and a test are used to determine whether an operation should be executed. Here is 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)- -; } } }
```

5 Experimental Results

We applied our algorithm to synthesize a software implementation of a real life embedded system: an ATM Server for Virtual Private Networks [2]. 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 8 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 FCPN containing 49 transitions and 41 places, of which 11 non-deterministic choices [8]. From the FCPN 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

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

Table I

implementations: the first, named QSS, was obtained using our Quasi-Static Scheduling technique presented in this paper 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 8. The results, obtained using a testbench of 50 ATM 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.

6 Conclusions

In this paper we have proposed a software synthesis technique based on quasi-static scheduling of Free Choice Petri Nets. The input of our tool is a FCPN model of the system to be designed, the output is a software implementation in C code composed of a number of tasks that are invoked at runtime by the RTOS. The C code implementation of each task is synthesized directly from a valid schedule of

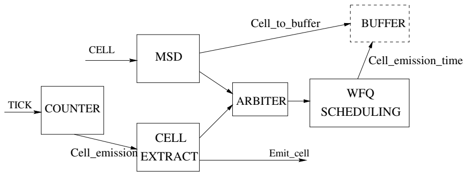

Figure 8: ATM Server example

the FCPN. We have presented an algorithm that first checks schedulability of the net and then, if the net is schedulable, computes a valid schedule, composed of as many cyclic firing sequences as the number of possible resolution of the non-deterministic choices. In the future it could also allow one to explore different schedules, evaluating tradeoffs between code and buffer size. Finally, we have described the application of our technique to a real case study and presented experimental results that clearly show that performances can be improved using our approach. The advantages of using our technique can be shortly outlined as follows. (1) Quasi-Static Scheduling, if compared to dynamic scheduling, minimizes the execution runtime overhead since it maximizes the amount of work done at compile time. (2) The model of computation is FCPNs, where schedulability is decidable. Differently from other algorithms applied to models (like BDF) that are undecidable, our algorithm is complete, in that it can solve the scheduling problem for any PN that is quasi-statically schedulable. (3) Schedulability of the specification is checked before the code is synthesized, differently from most existing approaches that assume schedulability a priori. (4) System functions with the same execution rate are automatically partitioned into a minimum number of concurrent tasks that are invoked at runtime by the RTOS.

References

[1] E.A.Lee and D.G.Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Transactions on computers*, January 1987.

[2] E.Filippi et al. Intellectual property re-use in embedded system co-design: an industrial case study. In *International Symposium System Synthesis*, December 1998.

[3] F. Thoen et al. Real-time multi-tasking in software synthesis for information processing systems. In *Proceedings of the International System Synthesis Symposium*, 1995.

[4] E.Teruel. *Structure theory of Weighted Place/Transition Net systems. The Equal Conflict hiatus.* Ph.D dissertation. Universidad de Zaragoza, 1994.

[5] J.Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model.* Ph.D dissertation. UC Berkeley, 1993.

[6] B. Lin. Software synthesis of process-based concurrent programs. In *Proceedings of the Design Automation Conference*, June 1998.

[7] M.Hack. *Analysis of Production Schemata by Petri Nets.* Master thesis. MIT, 1972.

[8] M. Sgroi. Quasi-static scheduling of embedded software using free-choice petri nets. Technical Report Memo No. UCB/ERL M98/, M.S. dissertation. UC Berkeley, May 1998.

[9] T.Murata. Petri nets: properties, analysis and applications. In *Proceedings of the IEEE*, April 1989.