Quasi-static scheduling for concurrent architectures

Jordi Cortadella
Universitat Politècnica de Catalunya
Barcelona, Spain

Alex Kondratyev
Cadence Berkeley Labs
Berkeley, USA

Luciano Lavagno
Politecnico di Torino
Torino, Italy

Yosinori Watanabe
Cadence Berkeley Labs
Berkeley, USA

Abstract

This paper presents a synthesis approach for reactive systems that aims at minimizing the overhead introduced by the operating system and the interaction among the concurrent tasks, while considering multiple concurrent execution resources. A formal model based on the notion of scheduling of Petri nets is used to perform the synthesis. We show how the notion of projections of a schedule for the complete system onto the components implemented on separate resources is essential to define the correctness of the partitioned schedule.

1 Introduction

Embedded systems use computers and electronics to perform some task, usually to control some physical system or to communicate information, without being explicitly perceived as a computer. Thanks to the ever-increasing performance at an ever-decreasing cost they are a preferred means to offer ever-improving services to a multitude of drivers, callers, photographers, watchers, and so on. The phenomenal growth of complexity and breadth of use of embedded systems can be managed only by providing designers with efficient methods for hardware or software synthesis, from formal models that explicitly represent the available concurrency. Software is becoming particularly interesting as an implementation option, due to the simultaneous growth of mask costs, which makes Application-Specific Integrated Circuits less appealing, and of CPU performance, which makes software a feasible choice even in presence of tight Real-Time constraints.

Concurrent specifications, such as dataflow networks [11], Kahn process networks [9], Communicating Sequential Processes [8], synchronous languages [6], and graphical state machines [7], are interesting because they expose the inherent parallelism in the application. However, their mixed hardware-software implementation on heterogeneous architectures requires to solve a fundamental *scheduling problem*. We assume in the following that the preliminary *allocation* problem of functional processes to architectural resources has been solved, either by hand or by some appropriate heuristic algorithm. The task of this paper is to define and solve the scheduling problem for a process-level concurrent functional specification allocated to *several computing resources*, in particular processors.

Most embedded systems are *reactive* in nature, meaning that they must process inputs from the environment at the speed and with the delay dictated *by the environment*. Scheduling of reactive systems thus is subject to two often contradicting goals: (1) satisfying timing constraints and (2) using the computing power without leaving the CPU idle for too long.

1.1 Static and Quasi-Static Scheduling

Static scheduling techniques do most of the work at compile-time, and are thus suitable for safety-critical applications, since the resulting software behavior is highly predictable [10] and the overhead due to task context switching is minimized. They may also achieve very high CPU utilization if the rate of arrival of inputs to be processed from the environment has *predictable regular rates* that are reasonably known at compile time.

Static scheduling, however, is limited to specifications without choice (Marked Graphs or Static Dataflow [11]). Researchers have recently started looking into ways of computing a static execution order for operations as much as possible, while leaving data-dependent choices at run-time. This body of work is known as *Quasi-Static Scheduling* (QSS) [2, 12, 13, 3, 14]. The QSS problem, i.e. the existence of a sequential order of execution that ensures no buffer overflow, has been proven to be undecidable by [2]

Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE

for specifications with data-dependent choices. Our work fits in the framework proposed by [3], in which Petri nets (PNs) are used as an abstract model, that hides away correlations among choices due to the value of data that are being passed around. We improve over [3] because we now consider *several execution resources*, and thus produce a *concurrent schedule* that exploits the available parallelism in both the specification and the implementation platform.

We use a game-theoretic intuitive formulation of the schedulability problem, in which the scheduler must win, by avoiding overflow of FIFO queues, against an adversary who can choose the outcome of non-deterministic data-dependent choices. The *scheduler* can resolve concurrency in an arbitrary, resource-dependent, fashion using a policy called *"schedule"* in the following, but it is not allowed to "starve" any input by indefinitely refusing to service it.

With respect to classical real-time scheduling theory, we focus on the *control and data dependencies* between processes, and create tasks based on them. I.e., two *fragments* of processes allocated to the same resource (e.g., a CPU) and whose execution is triggered by the same input from the environment are merged into the same task, in order to reduce inter-process communication and synchronization overhead. We allow *splitting and duplication* of process code, in order to come up with an efficient grouping of code fragments into tasks. Classical real-time scheduling theory can then be used to coordinate these tasks at run-time.

Further work will need to be devoted to the issue of optimal allocation in order to satisfy real-time constraints. In case inter-task scheduling is non-preemptive, the level of granularity at which processes can be merged also affects the overall schedulability.

1.2 Specification model

We consider a system to be specified as a set of concurrent processes. A set of input and output ports are defined for each process, and point-to-point communication between processes occurs through uni-directional FIFO queues between ports. Multi-rate communication is supported, i.e. the number of objects read or written by a process at any given time may be an arbitrary constant.

Communication operations on ports, as well as internal computation operations are modeled by *transitions* in the corresponding Petri net, while places are used to represent both sequencing within processes (a single token models the program counter) and FIFO communication (the tokens model the presence of the data items, while hiding their values).

Figure 1 depicts the specification of a concurrent system with a single master and two slaves, where the *Master* process reads an input from the port `IN` and then sends a request to one of the *Slave* processes. For communication

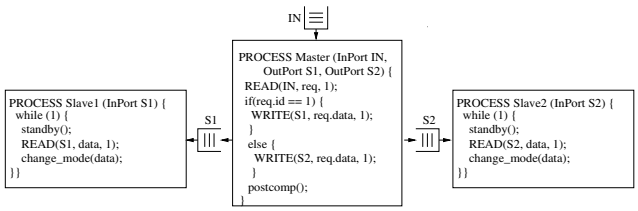
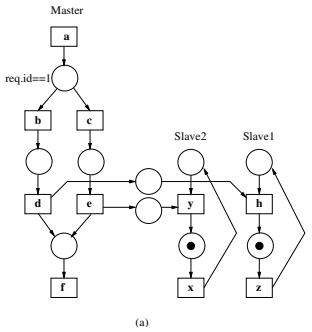


Figure 1. Specification for master-slave system



Transition	Operation
a	READ(IN, req, 1);
d	WRITE(S1, req.data, 1);
e	WRITE(S2, req.data, 1);
f	postcomp();
x	standby();
y	READ(S2, data, 1); change_mode(data);
z	standby();
h	READ(S1, data, 1); change_mode(data);

(b)

Figure 2. (a) Petri net model for the specification of Figure 1 (b) Operations associated with transitions

between processes, we support three types of operations: `READ`, `WRITE`, and `SELECT`. `READ(port, data, rate)` specifies an operation of reading data from the port `port` to a variable `data`, where the number of items read at a time is given by a constant integer `rate`. `WRITE` is similar, while `SELECT(port1, port2)` supports synchronization-dependent control, where it probes the presence of objects at the ports and non-deterministically selects one port with objects being available (See Figure 5(c) for example). Figure 2 shows a Petri net that models this specification.

2 Background

The following definitions introduce the nomenclature used in the paper.

Definition 1 (Petri net) *A Petri net is a 4-tuple* $N = (P, T, F, M_0)$*, where* P *is the set of places,* T *is the set of transitions,* $F : (P \times T) \cup (T \times P) \rightarrow \mathcal{N}$ *is the flow relation and* $M_0 : P \rightarrow \mathcal{N}$ *is the initial marking. The set of reachable markings of a Petri net is denoted by* $[M_0\rangle$*. The fact that* M' *is reachable from* M *by firing transition* t *is denoted by* $M[t\rangle M'$*. The pre-set and post-set of a node* $x \in P \cup T$ *are denoted by* ${}^\bullet x$ *and* $x^\bullet$*, respectively.*

Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE

Given a Petri net N with $P = (p_1, \ldots, p_n)$, the notation $\mathsf{Pre}[t]$ is used to represent the vector $(F(p_1,t), \ldots, F(p_n,t))$. Given a set of nodes X, $N \setminus \{X\}$ denotes the subnet of N obtained by removing the nodes in X and their adjacent arcs from N. If for any node x in PN N we have ${}^\bullet x \cap x^\bullet = \emptyset$, then N is called self-loop free. $M(p)$ denotes a number of tokens in place p under marking M.

In this paper we use nets with *source* transitions, i.e. with empty pre-sets. These transitions model the behavior of the input stimuli to a reactive system.

Definition 2 (Source and non-source transitions) *The set of transitions of a Petri net is partitioned into two subsets as follows:*

$$T_S = \{t \in T \mid {}^\bullet t = \emptyset\}, \;\; T_N = T \setminus T_S.$$

T_S and T_N are the sets of source *and* non-source *transitions, respectively. The set of source transitions T_S is further partitioned into* controllable T_S^c *and* uncontrollable T_S^u *($T_S^u = T_S \setminus T_S^c$) transitions.*

Informally, the decision on firing controllable transitions belongs to the scheduler, while the firing of uncontrollable transitions is governed by the environment and is out of scheduler control. This aspect is elaborated in more detail in Section 3, when we introduce the definition of schedule.

Definition 3 (Free-choice set) *For non-source transitions T_N a* Free-choice Set *(FCS) is defined as a maximal subset of transitions C such that*

$$\forall t_1, t_2 \in C \text{ s.t. } t_1 \neq t_2 \text{ and } t_1, t_2 \in T_N :$$

$$\mathit{Pre}[t_1] = \mathit{Pre}[t_2] \;\wedge\; ({}^\bullet t_1)^\bullet = ({}^\bullet t_2)^\bullet.$$

Transitions from one FCS set are always enabled simultaneously. Firing of one of them disables the rest in case of a safe net. This is a convenient mean to express a fully non-deterministic behavior. We will call $\mathsf{FCS}(t)$ the set of transitions that belong to the same FCS of t. Any conflict inside a FCS is said to be free-choice. In Section 3 the notion of FCS is further extended to source transitions.

Definition 4 (Transition system) *A transition system is a 4-tuple $A = (S, \Sigma, \rightarrow, s_{in})$, where S is a set of states, Σ is an alphabet of symbols, $\rightarrow \;\subseteq S \times \Sigma \times S$ is the transition relation and s_{in} is the initial state.*

With an abuse of notation, we denote by $s \xrightarrow{e} s'$, $s \rightarrow s'$, $s \rightarrow, \rightarrow s, \ldots$, different facts about the existence of a transition with certain properties.

A path p in a transition system is a sequence of transitions $s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} s_3 \rightarrow \cdots \rightarrow s_n \xrightarrow{e_n} s_{n+1}$, such that the target state of each transition is the source state of the next transition and e_i denote firing events. A path with multiple transitions can also be denoted by $s \xrightarrow{\sigma} s'$, where σ is the sequence of symbols in the path.

For a finer look at the internal structure of TS it is helpful to distinguish firing regions of TS events.

Definition 5 (Firing region) *Given a transition system $A = (S, \Sigma, \rightarrow, s_{in})$, the* firing region *of an event $e \in \Sigma$, denoted by $FR(e)$, is the set of states $\{s \mid s \xrightarrow{e}\}$.*

Definition 6 (Entry border) *The entry border of a set of states S' is a subset of S' defined as follows:*

$$EB(S') = \{s' \in S' \mid \exists s \notin S' : s \rightarrow s'\}.$$

The entry border of a firing region is the set of states by which this region is entered in TS from outside.

In the suggested scheduling approach there is a close relationship between modeling the original system by a PN and a corresponding TS specifying the system schedule.

Definition 7 (TS conforming to a PN) *Given a PN $N = (P, T, F, M_0)$ a TS $A = (S, \Sigma, \rightarrow, s_{in})$ is said to be conforming to N iff the following conditions are met:*

1. $\Sigma = T$
2. *There is a mapping $\mu : S \rightarrow [M_0\rangle$, with $\mu(s_{in}) = M_0$.*
3. *If transition t is fireable in state s, with $s \xrightarrow{t} s'$, then $\mu(s)[t\rangle\mu(s')$ in N.*

Note that in the reachability graph of a PN there is no distinction between fireability and enabling because according to the PN semantics any enabled transition might fire. A TS conforming to a PN is introduced as a subset of the reachability graph in which the enabling of events coincides with those in the PN (t is enabled in s when it is enabled in $\mu(s)$) but their fireability might differ. This feature, as shown in Section 3, allows a scheduler to control the firing of system transitions by delaying them to the benefit of deriving an efficient schedule.

3 Sequential schedule

Scheduling of a PN imposes the existence of an additional control mechanism for the firing of enabled transitions. For every marking, a scheduler defines the set of *fireable* transitions as a subset of the enabled transitions. The composite system (PN+scheduler) proceeds from state to state by firing fireable transitions.

The following definition is an extension of [4] to take into account the difference between controllable and uncontrollable transitions.

Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE

Definition 8 (Sequential schedule) *Given a Petri net* $N = (P, T, F, M_0)$ *and a partition* $FC(T_S^u) = \{T_S^u\}$, *a* sequential schedule *of* N *is a transition system* $Sch = (S, T, \rightarrow, s_0)$ *with the following properties:*

1. Sch *is conforming to* N *and has a finite set of states* S.
2. *If* t_1 *is fireable in* s, *then* t_2 *is fireable in* s *if and only if* $t_2 \in$ FCS(t_1).
3. *For each state* $s \in S$, *there is a path* $s \stackrel{\sigma}{\rightarrow} s' \stackrel{t}{\rightarrow}$ *for each* $t \in T_S^u$.

In order for this definition to be consistent, the notion of FCS is extended to source transitions. We assume that for controllable transitions FCSs are defined dynamically by the scheduler. These transitions can be fired arbitrarily (in conflict or not), because their firing is completely under scheduler control. Formally, given the set S of states of a schedule, the FCS for the set of controllable transitions T_S^c is defined as a mapping $H : S \rightarrow 2^{T_S^c}$, such that for each $s \in S$, $H(s)$ must be enabled $\mu(s)$.

For uncontrollable transitions, an FCS is defined as a partition $FC(T_S^u) = \{T_1, \ldots, T_k\}$ that is imposed on T_S^u. In particular $FC(T_S^u)$ could be the whole set of transitions T_S^u. This partition is included in the system specification.

Property 1 of Definition 8 implies that the set of traces of Sch is contained into that of N (any feasible trace in the schedule is feasible in the original PN). Property 2 indicates that one FCS is scheduled at each state. Finally, property 3 denotes the fact that any input event from the environment will be eventually served.

Given a sequential schedule, a state s is said to be an *await state* if all uncontrollable source transitions belonging to an FCS are fireable in s. An await state models a situation in which the system is "sleeping" and waiting for the environment to produce an event.

Intuitively, scheduling can be deemed as a game between the scheduler and the environment. The rules of the game are the following:

- The environment makes a first move by firing any of the source transitions.
- The scheduler might pick up any of the enabled transitions to fire (property 2) with two exceptions:
  (a) it has no control over choosing which of the source transitions to fire and
  (b) it cannot resolve choice for data-dependent constructs (which are described by free-choice sets).

  In cases (a) and (b) the scheduler must explore all possible branches during the traversal of the reachability space, i.e. fire all the transitions from the same FCS. However it can decide the moment for serving the source transitions or for resolving a free-choice, because it can *finitely* postpone these by choosing some other enabled transitions to fire.

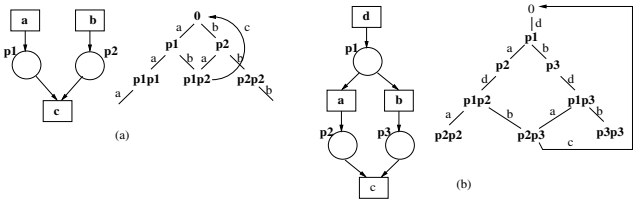


Figure 3. Non-schedulable PNs

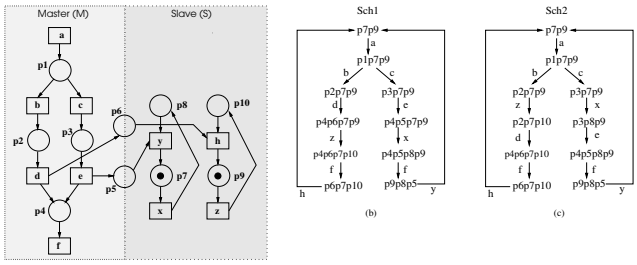


Figure 4. Master-slave system.

The goal of the game is to process any input from the environment (property 3) while keeping the traversed space finite (property 1). In case of success the result is to both classify the original PN as schedulable and derive the set of states (schedule) that the scheduler can visit while serving an arbitrary mix of source transitions. Under the assumption that the environment is sufficiently slow, the schedule is an upper approximation of the set of states visited during real-time operation.

The notion of sequential schedule is illustrated in Figures 3 and 4. Figure 3 shows two non-schedulable specifications and parts of their reachability spaces. The impossibility to find a schedule for the PN in Fig. 3(a) stems from the inability of a scheduler to control the firing of source transitions. A cyclic behavior in this PN is possible only with correlated input rates of transitions a and b. On the other hand, the PN in Fig. 3(b) is non-schedulable because of the lack of control on the outcome of free-choice resolution for the place $p1$.

Figure 4(a) presents an example of a master-slave system in which the master is non-deterministically choosing which one of the two slaves to trigger. The two possible schedules for this specification are given in Fig. 4(b)(c). These schedules show different interleavings of master and slave transitions.

Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE

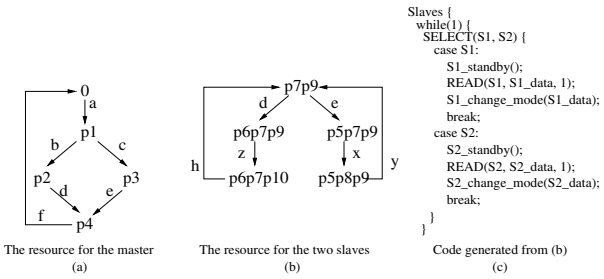


Figure 5. Schedules for the *Master* and the two *Slaves*

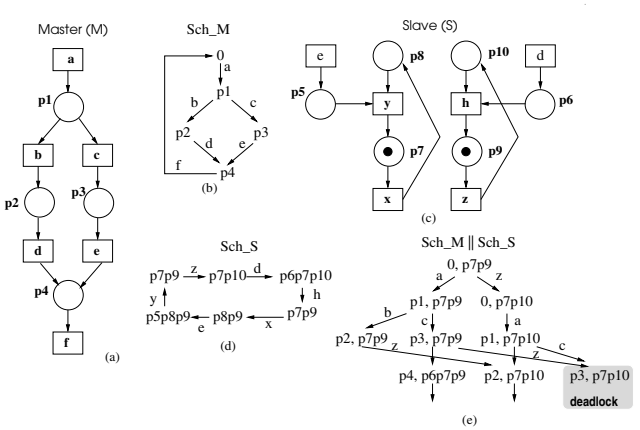


Figure 6. Schedules whose interaction leads to a deadlock

4 Concurrent schedules

4.1 Problem Overview

To address the scheduling problem with multiple resources for implementation, we assume that an allocation has been already determined, and we take it as input in addition to the Petri net for the specification. Intuitively, an allocation can be considered as a mapping from each transition to the resource that executes the operations represented by the transition. For example, in the Petri net shown in Figure 2, all the operations of the two *Slave* processes may be allocated to one resource, while those of the *Master* process may be allocated to another resource. In practice, we employ some restrictions on allocations, as formally defined in Section 4.2.

Given a Petri net and its allocation, the problem is to find a sequential schedule for the operations allocated to each resource. In the Master-Slave example, one may obtain the schedules given in Figure 5(a) and (b) for the resources for the *Master* and the two *Slave*s respectively. Figure 5(c) depicts the code generated from the schedule of Figure 5(b).

A naive approach for this scheduling problem is to compute a sequential schedule for each resource independently. However, this approach often results in a deadlock when the schedules are executed altogether. This problem can be illustrated as follows. In Master-Slave example, the allocation given above defines two Petri net fragments, one for the *Master* and the other for the two *Slave*s, as shown in Figure 6(a) and (c) respectively. Note that the Petri net for the *Slave*s has transitions d and e as source transitions, even though their operations are allocated to the resource for the *Master*. This is because the executions of these operations need to be taken as input in order to define the behavior of the *Slave*s.

The naive approach will take these Petri nets, and find a sequential schedule for each. For the Petri net representing the *Slave*s, one needs to decide whether the source transitions d and e should be treated as controllable or uncontrollable, without knowing the behavior of the *Master* process. Treating them as uncontrollable is not good in general, since it works only when their operations are in conflict in the *Master*. Once they are treated as controllable, one needs to decide when and how they are fired in the schedule for the *Slave*s. Suppose that the schedules shown in Figure 6(b) and (d) have been obtained for the *Master* and the *Slave*s. How these schedules interact when executed in the two resources can be identified by taking the parallel composition of the two, as partially shown in Figure 6(e). As shown in the figure, a deadlock can result for these schedules, if for example the resource for the *Slave*s executes the transition z while the resource for the *Master* executes a and c. In the original specification shown in Figure 2, if a and c are executed, then transition e has to be executed. However, the schedule for the *Slave*s treated e as a controllable source, and it can be executed only after h, according to the schedule of Figure 6(d). Since h requires d to be executed in the specification and since d is in conflict with a, the deadlock results. This kind of causality relation between the *Master* and *Slave* processes cannot be identified when schedules are computed independently for the resources, and thus the naive approach works only when it by chance finds *correct* schedules for all the resources. In the next sections, we show conditions under which schedules for the resources do not cause this problem, and present a procedure that finds schedules accounting for these conditions.

4.2 Allocation

In [3] it was shown that the main advantage of the implementation obtained by sequential QSS with respect to the one directly implementing a set of concurrent processes is a drastic decrease of the communication overhead. For the case of a single computational resource (e.g. a CPU), sequential QSS gives an optimal solution. However when several computational resources are available the sequential implementation might result in a significant performance penalty. This motivates an investigation of concur-

Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE

rent schedules.

When several computational resources are available, the actions of the original specification (PN transitions) must be assigned to resources for implementation. In quasi-static scheduling this assignment is done statically and is formalized through the notion of allocation. A concurrent schedule is defined with respect to a given allocation, and the problem of finding an optimal allocation is left to future work.

Definition 9 (Allocation) *Given a Petri net* $N = (P, T, F, M_0)$ *an allocation defines a partition of* T, $Alloc(T) : T \rightarrow \{1, 2, \ldots, n\}$ *(where each integer between 1 and* n *denotes a resource) with the following properties:*

1. $\forall p,\ \forall t_1, t_2 \in {\bullet}p \ \Rightarrow \ Alloc(t_1) = Alloc(t_2)$
2. $\forall p,\ \forall t_1, t_2 \in p{\bullet} \ \Rightarrow \ Alloc(t_1) = Alloc(t_2)$
3. $\forall p,\ [(t_1 \in {\bullet}p) \ \wedge \ (t_2 \in p{\bullet}) \ \wedge \ Alloc(t_1) \neq Alloc(t_2)] \ \Rightarrow \ FCS(t_1) = \{t_1\}.$

Transitions with the same allocation value are meant to be implemented by a single resource. This uniquely defines an FC partition of uncontrolled transitions: a source partition is called *allocation matching* when $\forall t_i, t_j \in T_S^u$, t_i, t_j belongs to the same FCS if and only if $Alloc(t_i) = Alloc(t_j)$.

Note that for non-source transitions allocation preserves FCS partitioning because all the output transitions of the same choice place must be put in the same allocation cluster. The places between transitions with different allocation values are interpreted as port places and are used for resource interfacing. Property 1 and 2 of allocation guarantee that writing to (reading from) port places could be done by transitions allocated to the same resource only, while Property 3 tells that writing is always done in a deterministic way that ensures a separation between making non-deterministic choice and performing communication.

It is easy to see that for a PN derived from a set of concurrent processes (see Section 1) any allocation that respects process boundaries (i.e. all transitions of the same process are assigned the same allocation value) satisfies Properties 1-2. To satisfy Property 3 one might need to introduce silent transitions to decouple choice and communication. The latter is always possible and is known to be an equivalent transformation.

In that way for the suggested specification style an allocation could execute several processes on single resource, but it never splits processes between several resources.

Definition 10 *Given a Petri net* $N = (P, T, F, M_0)$ *with an allocation* $Alloc(T)$*, an allocation cluster* i *is a PN subnet defined by a subset of transitions* $T_a = T_{internal} \cup T_{input}$*, where* $t \in T_{internal} \iff Alloc(t) = i$*, while* $t \in T_{input} \iff (t \in {\bullet}({\bullet}(t')) \wedge Alloc(t') = i)$ *and subset of places* $P_a = {\bullet}(T_{internal})$.

I.e. a cluster contains transitions with the same allocation value and their immediate predecessors and places that are input to its internal transitions.

Figure 4(a) shows an allocation for the Master-Slave example that naturally partitions $Master$ and $Slave$ functionality on different resources. It is easy to check that this allocation satisfies Properties 1-3 of Definition 9.

The clusters for $Master$ and $Slave$ corresponding to the allocation in Figure 4(a) are shown in Figure 6(a) and (c) respectively. Note that clusters are overlapping by input transitions of port places (transitions d and e for $Slave$).

4.3 Definition of concurrent schedule

The game-theoretic interpretation of scheduling discussed in Section 1 can be extended to concurrent scheduling. However, the rules of the game must be extended to take care about the proper composition of distributed parts of the scheduler implementation, since the global scheduler is indeed a composition of one local scheduler per resource.

These extensions concern two main issues 1) the commitment to decisions about transition fireability and 2) the receptiveness to environment inputs.

Definition 11 (Persistent firing region) *Given TS* $A = (S, \Sigma, \rightarrow, s_{in})$ *conforming to PN* N, $FR(t)$ *is called persistent in* A *if* $\forall s \in FR(t)$ *such that* $s \stackrel{t'}{\rightarrow} s', s' \notin FR(t)$ *either 1)* $t' = t$ *or 2)* t *becomes disabled in* s' *or 3) transitions* t *and* t' *belong to the same FCS.*

Note that cases 2 and 3 are different. On one hand, transitions of non free-choice PNs can disable each other and not be in the same FCS (i.e. Case 3 does not cover Case 2). On the other hand, source transitions cannot be disabled, but can belong to the same FCS (i.e. Case 2 does not cover Case 3).

Informally, one can exit from a firing region for t either by firing t or by disabling t through the firing of some other transition t' which is in conflict with t. For source transitions the disabling is interpreted in a broader sense as containment in the same FCS (see Condition 3).

Persistency helps to formulate the commitment of the distributed scheduler to the decisions about transition firings and makes it impossible for a scheduler to "withdraw its moves" when playing against the environment.

Another important requirement is the receptiveness of a schedule. It describes the ability to make progress under any input generated by the environment. Receptiveness of a sequential schedule is guaranteed by forcing the firing of all source transitions once any of them becomes fireable (in await states). For a concurrent schedule it would

Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE

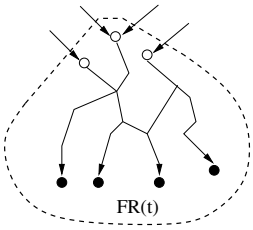


Figure 7. Receptive firing region.

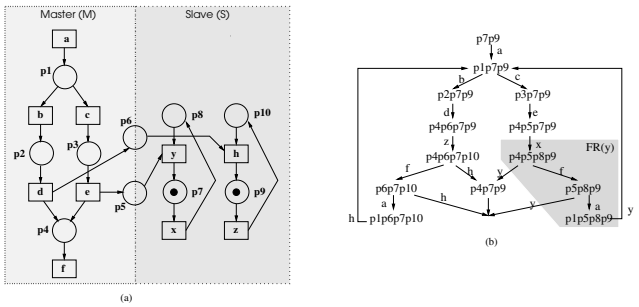


Figure 8. Concurrent schedule for master-slave specification.

be too restrictive to synchronize all source transition firings in a single state. For efficient operation, processes implemented by different resources must be able to move faster or slower with respect to their neighbors. Therefore it is possible that, due to the difference of speed among processes, some source transitions become enabled earlier than others. Then if the environment produces inputs for faster processes at a faster rate, these processes might benefit from that by not waiting for the rest of the system to catch up at a common synchronization point. However, this is not a hard requirement for the environment, which still behaves non-deterministically and produces the inputs at will. In order to guarantee progress for any input combination, the relaxed receptiveness property below states that every time a source transition t becomes enabled, it is still possible to reach an await state through the firing of non-source transitions concurrently fireable with t. This ensures that the schedules cannot favor some of the uncontrollable input transitions with respect to others.

Definition 12 (Receptive firing region) *The firing region $FR(t)$ of source transition t is called receptive if for every state s from the entry border of $FR(t)$ any maximal trace* [1] *$\sigma : s \stackrel{\sigma}{\rightarrow}$ not containing source transitions and not leaving $FR(t)$ ends up in an await state within $FR(t)$.*

Definition 12 states that once a firing region of some source transition t is entered and the firing of source transitions (including t) is postponed by the scheduler, then sooner or later a state with all source transitions being fireable (await state) is reached (see Figure 7).

Allocation, receptiveness and persistency are the new features (with respect to the sequential case) that one needs to consider in defining concurrent schedules.

Definition 13 (Concurrent schedule) *Given a Petri net $N = (P, T, F, M_0)$, allocation $Alloc(N)$ and partition $FC(T_S^u)$ matching it, a* concurrent schedule *of N is a transition system $Sch = (S, T, \rightarrow, s_0)$ with the following properties:*

1. *Sch is conforming to N and has a finite set of states S.*
2. *If t_1 is fireable in s, then any $t_2 \in$ FCS(t_1) is fireable in s as well, while any $t_2 \notin$ FCS(t_1) with $Alloc(t_1) = Alloc(t_2)$ is not fireable in s.*
3. *All firing regions are persistent.*
4. *All firing regions of uncontrollable transitions are receptive.*
5. *For each state $s \in S$, there is a path $s \stackrel{\sigma}{\rightarrow} s' \stackrel{t}{\rightarrow}$ for each $t \in T_S^u$.*

The need for Properties 3 and 4 in defining concurrent schedules has been discussed already. Property 2 is an extension of the similar requirement in sequential schedules. It tells that all transitions from the same FCS must be fireable simultaneously. Moreover it tells that at most one FCS from a cluster can be fireable in a schedule state. This implies that every cluster is implemented sequentially.

Figure 8(b) shows a concurrent schedule for the master-slave example. One can easily check that it satisfies Properties 1- 5. The shadowed area corresponds to a persistent firing region for transition y.

4.4 Construction of concurrent schedule

A concurrent schedule provides a global view on the behavior of all resources used in allocation. Ideally such a view should be derived as a composition of local schedules: one per resource (cluster). This strategy however meets some difficulties that were discussed in Section 4.1. It was shown there that an independent scheduling of each allocation cluster does not ensure the consistency of firing read/write transitions that produce/consume data in port places. To guarantee consistency, we suggest first to construct a sequential schedule for the whole system. A sequential schedule provides a uniform starting point for deriving schedules for clusters. Cluster schedules are obtained by projecting the global sequential schedule on the set of cluster transitions. This design flow is illustrated in Figure 9.

[1] σ is maximal with respect to property P if for any trace σ' extending σ ($\sigma' = \sigma a$), P is violated.

Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE

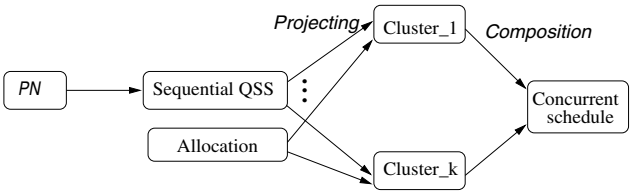


Figure 9. Concurrent scheduling flow.

A schedule projection is defined as a sequence of transformations of the underlying TS. The following notation is introduced to describe them:

1. $Pred(s) = \{s_i \mid s_i \xrightarrow{e} s\}$ (set of immediate predecessors of s)
2. $Succ(s) = \{s_i \mid s \xrightarrow{e} s_i\}$ (set of immediate successors of s)

Definition 14 (State merge) *Given a TS $A = (S, \Sigma, \rightarrow, s_{in})$ and a pair of states $s_1, s_2 \in S$, the merge of s_1, s_2 results in a new TS $A' = (S', \Sigma', \rightarrow', s'_{in})$ such that $S' = (S \cup \{s_{12}\}) \setminus \{s_1, s_2\}$ and $Pred(s_{12}) = (Pred(s_1) \cup Pred(s_2)) \setminus \{s_1, s_2\}$, $Succ(s_{12}) = (Succ(s_1) \cup Succ(s_2)) \setminus \{s_1, s_2\}$, while for the rest of states in A' the sets of their predecessors and successors coincide with those in A.*

Merging a pair of states s_1, s_2 replaces these states by a single state s_{12} which combines immediate predecessors and successors from s_1 and s_2.

Definition 15 (TS projection) *Given a TS $A = (S, \Sigma, \rightarrow, s_{in})$ and a set of events $E \subseteq \Sigma$, the projection of A on E is a TS obtained by merging every pair of states s_1, s_2 such that $s_1 \xrightarrow{e} s_2 \wedge (e \notin E)$.*

Proposition 1 *For a given TS $A = (S, \Sigma, \rightarrow, s_{in})$ and set of events $E \subseteq \Sigma$, the projection of A on E is unique.*

The proof immediately follows from the commutativity of the $\cup$ operation.

Definition 16 (Deterministic TS) *A TS $A = (S, \Sigma, \rightarrow, s_{in})$ is deterministic iff*

1. $\forall s \xrightarrow{e_1} s_1, s \xrightarrow{e_2} s_2 : [e_1 = e_2 \Rightarrow s_1 = s_2]$.

2. $\forall s_1 \xrightarrow{e_1} s, s_2 \xrightarrow{e_2} s : [e_1 = e_2 \Rightarrow s_1 = s_2]$.

Definition 17 (Determinization of TS) *The determinization of a TS $A = (S, \Sigma, \rightarrow, s_{in})$ is the TS obtained from A as the fixed point in applying state merging for every pair of states $s_1 \neq s_2$ such that either*

1. $\exists s, e \mid s \xrightarrow{e} s_1, s \xrightarrow{e} s_2$ *or*

2. $\exists s, e \mid s_1 \xrightarrow{e} s, s_2 \xrightarrow{e} s$

Proposition 2 *The determinization of a TS is unique.*

The proof follows from the fact that the set of immediate successors and predecessors is monotonically increasing during state merging, and that the pre-conditions for merging are transitive. Therefore if a pair of states s_1 and s_2 are merged into a single state s_{12} then any other state s_3 that satisfies the conditions of determinization either in pair with s_1 or in pair with s_2, would clearly satisfy the determinization conditions for the pair $\{s_3, s_{12}\}$.

Proposition 2 proves the soundness of Definition 17 as it states the uniqueness of the fixed point during state merging.

Definition 18 (Schedule projection) *Given a schedule $Sch = (S, \Sigma, \rightarrow, s_{in})$ and a set of events $E \subseteq \Sigma$, the projection of Sch on E is the result of projection on E and determinization of the underlying TS.*

4.5 Consistent schedules

Definition 19 *Given a Petri net $N = (P, T, F, M_0)$ and clusters $C_1, \ldots, C_k$ defined by allocation $Alloc(N)$, the set of sequential schedules $Sch_{C_1}, \ldots, Sch_{C_k}$ of $C_1, \ldots, C_k$ is called consistent if their parallel composition is isomorphic to a concurrent schedule of N.*

Projecting a sequential schedule of PN N onto subsets of events of clusters $C_1, \ldots, C_k$ is the constructive way in which we would like to seek for a consistent set of schedules. However projecting a sequential schedule onto a set of events of a cluster does not always result in a valid sequential schedule for this cluster. To illustrate that, let us return to the Master-Slave example and its sequential schedule $Sch2$ from Figure 4(c).

The clusters for $Master$ and $Slave$ are shown in Figure 6(a)(c). The result of projecting $Sch2$ onto sets of events T_M and T_S of these clusters is illustrated by Figure 10.

A closer look at the slave projection $Proj(S)$ shows that, because of merging of states labeled with transitions a, b, c, f (these transitions do not belong to T_S), both z and x are fireable in the initial state of $Proj(S)$. However, in a sequential schedule only transitions from the same FCS could be fireable in a particular state. But z and x do not belong to the same FCS in the PN for Master-Slave. Section 4.2 points out that allocation must preserve FCSs. Therefore $Proj(S)$ is not a valid sequential schedule for $Slave$ because it fires transitions from different FCSs in the same marking.

The following Proposition gives a constructive way to check whether projections of a sequential schedule of the overall system result in a consistent set of schedules for its clusters.

Proposition 3 *Let $C_1, \ldots, C_k$ be a set of clusters of PN $N = (P, T, F, M_0)$ defined by allocation $Alloc(N)$ and let*

Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE

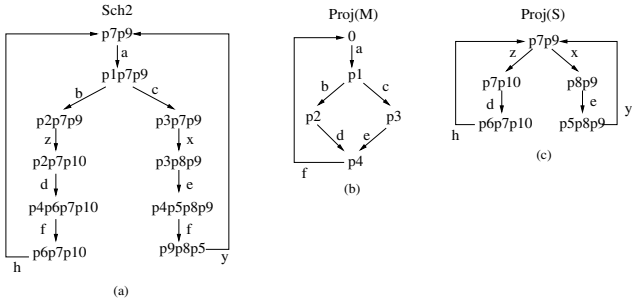


Figure 10. Inconsistent projections.

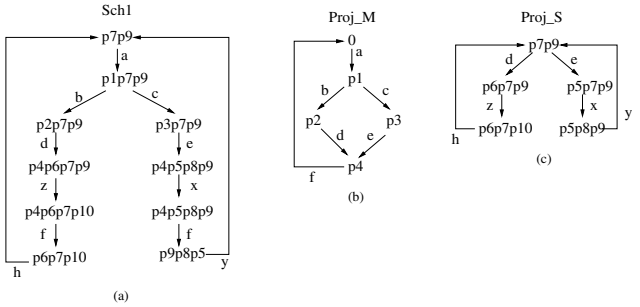


Figure 11. Consistent projections.

Sch be a sequential schedule of N. If the set of projections $PR = \{Proj(C_1), \ldots, Proj(C_k)\}$ of Sch onto events of $C_1, \ldots, C_k$ gives sequential schedules for $C_1, \ldots, C_k$, then PR is consistent.

See the Appendix for the proof.

One could construct many sequential schedules by the very same PN, with different event interleavings (deciding the order of event firing is the main part of a scheduler policy). Some of these schedules might serve better in deriving a set of consistent projections.

To illustrate this let us explore another sequential schedule $Sch1$ for the Master-Slave example (Figure 11(a)). $Sch1$ differs from $Sch2$ in Figure 10(a) by reversing the order of concurrent transitions d, z and e, x. Projections of $Sch1$ onto clusters for $Master$ and $Slave$ (Figure 11(b) and (c) respectively) give sequential schedules for these clusters and therefore present a consistent set of schedules. Contrary to Figure 10(c), in the initial state of Slave projection of Figure 11(c) only controllable transitions are fireable. This does not contradict the FCS relation because for controllable transitions FCSs are defined dynamically.

The capability of a sequential schedule to produce a consistent set of projections under the given allocation can be taken into account during the construction of a schedule. Let us assume for simplicity that the schedule is constructed in such a way that no two states of a schedule get the same marking. I.e., the schedule is minimized on the fly by merging the states with the same markings[2]. Let us associate with every state of the schedule not only the corresponding PN marking, but also the local markings for each allocation cluster, obtained as projections of PN markings onto the subset of places defined by the cluster. Every time a new schedule state s is generated ($s =< M, m_1, ..., m_n >$, where M is a PN marking and $m_1, ..., m_n$ are local markings for clusters 1,...,n) it must pass the consistency check as follows:

- for each allocation cluster $C_i, i = 1, \ldots, n$ **do**
  - In the currently obtained set of schedule states find the subset $S(m_i)$ in which all local markings for C_i coincide with m_i (these states would correspond to the same state in the projection for cluster C_i)
  - **If** the union of fireable transitions of states from $S(m_i)$ is not in the same FCS **then** *exit(failure)*
- *exit(success)*

If the check for consistency of the state s returns "success", the schedule continues with s, while in case of failure it backtracks and explores different ordering of transition firings. In that way the consistency check serves as an additional condition for termination.

The above procedure illustrates that algorithms used to generate a sequential schedule, e.g. the one in [3], need minor modifications to include that consistency check. It is possible that backtracking in the generation process happens more often than in the sequential case. This problem could be alleviated by developing heuristics and exploring sufficient conditions that simplify the consistency analysis, but this is left to future work.

5 Conclusions

This paper proposes a method that bridges the gap between specification and implementation of reactive systems. From a set of communicating processes, and by deriving an intermediate representation based on Petri nets, a set of concurrent tasks that serve input events with minimum communication effort is obtained. We extend previous work by considering a more general definition of the concept of schedule, considering concurrent implementations. This considerably increases the applicability of the method, but requires additional considerations in order to prove that tasks scheduled on different resources interact correctly and do not deadlock due to the partitioning.

[2]The case when several schedule states corresponds to the same marking could be treated in a similar way but requires some additional bookkeeping.

Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE

In the future, we would like to apply our technique to realistic examples, and consider the problem of allocating processes to resources in order to improve the performance of the resulting schedules, under cost and real-time constraints.

Acknowledgment

This work has been partially funded by a grant from Cadence Design Systems and CICYT TIC2001-2476.

References

[1] G. Arrigoni, L. Duchini, L. Lavagno, C. Passerone, and Y. Watanabe. False path elimination in quasi-static scheduling. In *Proceedings of the Design Automation and Test in Europe Conference*, March 2002.

[2] J. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, U.C. Berkeley, 1993.

[3] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. Sangiovanni-Vincentelli. Task Generation and Compile-Time Scheduling for Mixed Data-Control Embedded Software. In *Proceedings of the 37th Design Automation Conference*, June 2000.

[4] J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone, and Y. Watanabe. Quasi-static scheduling of independent tasks for reactive systems. In *Proceedings of the International Conference of Appliaction and Theory of Petri Nets*, 2002.

[5] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. YAPI: Application Modeling for Singal Processing Systems. In *Proceedings of the* 37^{th} *Design Automation Conference*, June 2000.

[6] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[7] D. Har'el, H. Lachover, A. Naamad, A. Pnueli, et al. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), April 1990.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 66 Wood Lane End, Hemel Hempstead, Hertfordshire, HP2 4RG, UK, 1985.

[9] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP Congress*, August 1974.

[10] H. Kopetz and G. Grunsteidl. TTP – A protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1), January 1994.

[11] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow graphs for digital signal processing. *IEEE Transactions on Computers*, January 1987.

[12] B. Lin. Software synthesis of process-based concurrent programs. In *35th ACM/IEEE Design Automation Conference*, June 1998.

[13] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice Petri nets. In *36th ACM/IEEE Design Automation Conference*, June 1999.

[14] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and et al. Scheduling hardware/software systems using symbolic techniques. In *International Workshop on Hardware/Software Codesign*, 1999.

6 Appendix

Before proving Proposition 3, let us introduce some additional notions and show the validity of an intermediate property.

Definition 20 (Projection image) *Given a* TS A' *obtained by projection and determinization of* TS $A = (S, \Sigma, \rightarrow, s_{in})$ *on events* $E \subset \Sigma$*, the image of a state* $s \in A'$ *is the set of states in* A *that are merged into* s *(denoted by* $im(s)$*).*

Property 1 *Let* $Proj(C)$ *be the projection of a sequential schedule* Sch *onto subset of events* T^C *of an allocation cluster* $C = (P^C, T^C, F^C, M_0^C)$*. Then for any state* $s' \in Proj(C)$ *the projections of* PN *markings onto* P^C *for all* Sch *states in the image of* s' *coincide.*

Proof: Let us consider the construction of $Proj(C)$ in two steps:

– $Proj1(C)$ as a result of state merging because of hiding events from $T \setminus T^C$ and

– $Proj2(C)$ as a result of determinization of $Proj1(C)$.

Let $s1, s2 \in Sch$ and $s1 \xrightarrow{t} s2$ where $t \notin T^C$. Then $s1$ and $s2$ have the same image $s' \in Proj(C)$ because they must be merged. Allocation defines a partition on the set of places P of the original PN. From this follows that t cannot change the marking of any of the place from P^C and hence projections of $\mu(s1)$ and $\mu(s2)$ onto P^C coincide (projection of a marking $\mu(s)$ on a subset of places P^C results in removing from $\mu(s)$ all the places that are not in P^C, this projection will be called local marking of cluster C). From

Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE

this follows that all states of Sch in the image of any state from $Proj1(C)$ have the same local markings.

Consider $s' \xrightarrow{t} s1'$ and $s' \xrightarrow{t} s2'$ in $Proj1(C)$, where $t \in T^C$. Then in Sch there exist two pairs of states $s3 \xrightarrow{t} s4$ and $s5 \xrightarrow{t} s6$ such that $s3$ and $s4$ belong to the images of s' and $s1'$, and $s5$ and $s6$ belong to the images of s' and $s2'$. Since $s3$ and $s5$ have the same image, their local markings coincide. From this follows that $\mu(s4)$ and $\mu(s6)$ have the same local markings because they are obtained as a result of the firing of the same transition t from the same local marking. Hence $\mu(s3) = \mu(s5)$. $\diamond$

Note. For the schedules in which no two schedule state have a same marking the projections could be derived by simply projecting Sch markings onto local markings and then merging all projection states that have the same local markings.

Proposition 3. *Let $C_1, \ldots, C_k$ be a set of clusters of PN $N = (P, T, F, M_0)$ defined by allocation $Alloc(N)$ and Sch be a sequential schedule of N. If the set of projections $PR = \{Proj(C_1), \ldots, Proj(C_k)\}$ of Sch onto events of $C_1, \ldots, C_k$ gives sequential schedules for $C_1, \ldots, C_k$ then PR is consistent.*

Proof: Let us show that the parallel composition of $C_1, \ldots, C_k$ (denoted by $||C_{1,k}$) satisfies Definition 13 of concurrent schedule.

• *Conformance to PN and finiteness (Condition 1 of Definition 13).*

The set S of states of a parallel composition is finite because each of the projections is finite. To show the conformance of $||C_{1,k}$ to PN N one needs to show the existence of a mapping from states of the schedule to PN markings and make sure that events are fired in a schedule state only when they are enabled in the corresponding marking (see Definition 7).

Let us construct the mapping $\mu : S \rightarrow [M_0\rangle$, with $\mu(s_0) = M_0$. μ is obtained through the union of local markings for states from S, i.e. for any $s \in S$ ($s =< s^1, ..., s^k >$, where $s^1, ..., s^k$ are states of $Proj(C_1), \ldots, Proj(C_k)$) the mapping μ provides a marking $\mu(s)$ which is obtained as a union of local markings of $s^1, ..., s^k$. Let us show that $\mu(s)$ corresponds to a PN marking in $[M_0\rangle$ and any fireable transition in s is enabled in $\mu(s)$.

Let us apply induction on the length of the path from s_0 to s.

$\mu(s_0) = M_0$ is trivial and directly follows from the rules of projection. Let a be fireable in s. Then for every cluster C_i such that $a \in C_i$, transition a must be fireable in the local state s^i (due to the rules of parallel composition). Allocation defines a partition and therefore the set of all clusters C_i such that $a \in C_i$ contains all input places of a in the original PN (with each place having sufficient number of tokens for enabling of a). Therefore if $s \xrightarrow{a}$ then a is enabled in $\mu(s)$. Moreover, the additivity of marking change implies that if $s \xrightarrow{a} s1$, then $\mu(s1)$ coincides with the marking obtained from $\mu(s)$ by the firing of a. Condition 1 is proved.

• *FCS fireability (Condition 2 of Definition 13).*

It t_1 fireable in a schedule state s of $||C_{1,k}$ then t_1 is also fireable in a local state s^i of cluster C_i containing t_1. Allocation preserves FCS and therefore any $t_2 \in$ FCS(t_1) is also fireable in s^i. Hence t_2 must be fireable in s as well. From $Proj(C_i)$ being a sequential schedule it follows that only one FCS from cluster C_i could be fireable in local state s^i. Therefore in schedule state s at most one FCS from each cluster is fireable.

• *Persistency (Condition 3 of Definition 13).*

Let us assume that persistency is violated in $||C_{1,k}$, i.e. $s \xrightarrow{a} s1$, $s \xrightarrow{b} s2$ and a is not fireable in $s2$. Clusters do not overlap on places and therefore events from different clusters could not disable each other. Then there must exist a local state s^i with $a, b \in C_i$ such that the firing of b disables a in C_i. Then b and a must be in the same FCS in $Proj(C_i)$. Hence a and b are in the same FCS in the original PN as well which satisfies persistency.

• *Receptiveness (Condition 4 of Definition 13).*

Let state $s =< s^1, ..., s^k >$ in $||C_{1,k}$ belong to the entry border of the firing region $FR(t)$ of uncontrolled source transition t and $t \in C_i$. Then t is fireable in state s^i of cluster C_i. According to projection rules the image of s^i in Sch_{seq} contains an await state s_a and all states from which s_a is reached by firing transitions that are not from C_i. Let us take state $s' \in Sch_{seq}$ which is in the entry border of $im(s^i)$ (see Figure 12). From s' there exists a trace δ into await state s_a which does not contain transitions from C_i or from T_S^u ($\delta = \delta1, \delta2, \delta3$ in Figure 12). By δ one can derive a parallel run $||\delta$ which contains all possible permutations of concurrent transitions in δ. s' is chose in such a way that s is in $||\delta$. Therefore in $||C_{1,k}$ there exists a trace from s to await state s_a that is covered by the set of traces $||\delta$. It is a maximal trace because it cannot be extended beyond s_a.

To prove that any maximal trace from s has the same property let us consider traces of Sch_{seq} whose parallel runs correspond to maximal traces from s. Let them start from s' but diverge with δ. Any such trace σ ($\sigma = \sigma1, \sigma2$ in Figure 12) either ends up in an await state (see Property 3 of Definition 8) or it has a loop from non-source events (trace w in Figure 12). In the latter case however the trace corresponding to the loop is not maximal.

• *Liveness with respect to uncontrollable transitions (Condition 5 of Definition 13).*

Instead of cyclic objects for the schedule and its projections, let us consider their acyclic infinite representations in the form of unfoldings

Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE

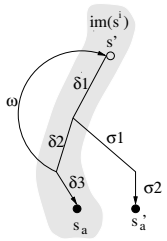


Figure 12. Proof of receptiveness.

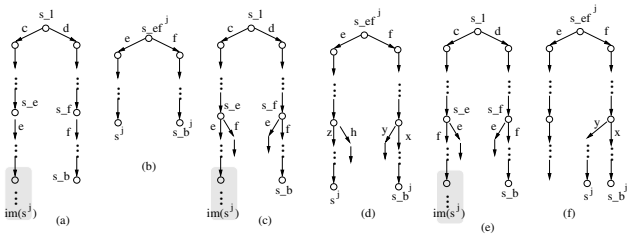


Figure 13. Proof of liveness.

$Sch^u_{seq}, Proj(C_1)^u, ..., Proj(C_k)^u, ||C^u_{1,k}$. Suppose that in $||C^u_{1,k}$ there exists a state $s =< s^1, ..., s^k >$ from which a source transition a is unreachable. Without loss of generality one can assume $a \in C_1$. Let us consider the composition $||C_{2,k}$ of projections $Proj(C_2), ..., Proj(C_k)$.

–Case 1. $||C_{2,k}$ is a concurrent schedule for the subnet of the original PN N obtained by deleting transitions from C_1 together with their output places.

Then the non-liveness of a stems from the inconsistency of $C_{2,k}$ and C_1. Transition a is unreachable in $||C^u_{1,k}$ from $s =< s^1, ..., s^k >$. However in $Proj(C_1)^u$ there must be a feasible sequence of fireable transitions $b, c, ... \mid s^1 \xrightarrow{b} s^1_2 \xrightarrow{c} ...s^1_i \xrightarrow{a}$ because according to conditions of Proposition 3, $Proj(C_1)$ is a sequential schedule for C_1. Let state s be chosen in such a way that b is not fireable in s (if b is fireable in s then we will consider state $s' \mid s \xrightarrow{b} s'$ and the next transition c in the sequence from s^1 to s^1_i). Then b is not fireable in state s only if s contains local state $s^j \mid s^j \in Proj(C_j)^u$, where $b \in C_j$ and b is not fireable in s^j.

– Case 1.1. b is not reachable in $Proj(C_j)^u$.

In Sch^u_{seq} let us consider state $s_b \in im(s^1) \mid s_b \xrightarrow{b}$ and a set of states $im(s^j)$. None of the states from $im(s^j)$ can precede s_b, otherwise b would be reachable from s^j in $Proj(C_j)^u$. State s_b cannot precede states from $im(s^j)$ by the choice of s^j (otherwise b cannot be blocked in s). Let us choose the last state s_l in Sch^u_{seq} such that $s_l \xrightarrow{\sigma} s_b$ and $\exists s \in im(s^j) \mid s_l \xrightarrow{\gamma} s$ (i.e. from any successor of s_l it is not possible to reach both s_b and $im(s^j)$. Then in state s_l at least two events c and d are fireable, where c is the first event in σ, while d is the first event in γ (see Figure 13(a)).

Events c and d are from the same FCS and due to this are internal for some cluster. Then they are non-observable for at least one of the clusters C_j or C_1 (say C_j e.g.). Let us consider events e and f, $e, f, \in C_j$ that are first met in σ and γ. In the projection of Sch^u_{seq} onto events from C_j the corresponding states s_e and s_f would be merged into s_{ef} and e and f would be fireable from s_{ef} (see Figure 13(b)). e and f are from the same FCS because otherwise $Proj(C_j)$ cannot be a valid sequential schedule. Then they both must be fireable in states s_e and s_f of Sch^u_{seq}. Two cases are possible:

a) s_b and $im(s^j)$ are reachable by making different choices for e and f (see Figure 13(c)) and

b) s_b and $im(s^j)$ are reachable by making the same choice for e and f (see Figure 13(e))

In case (a) due to the choice of s_l none of the states from $im(s^j)$ are reachable from state s_f. Because of this in the $Proj(C_j)^u$ once event e is fired, there must exist another forking point (denoted in Figure 13(d) by z and h) to make $im(s^j)$ unreachable from s_f. For z and h it is possible to repeat the same consideration as for e and f, with the exception that these events are closer to s^j. Finally we will either arrive to the contradiction of keeping FCS relations in projections (like in Figure 13(b)) or will exhaustively check all the forks in σ or γ without distinguishing conditions on reachability of $im(s^j)$ and s_b. The latter also leads to contradiction and proves that the assumption about the validity of case (a) is wrong.

Case (b) (Figure 13(e)) reduces to case (a) because the need to distinguish the reachability conditions for s_b and $im(s^j)$ requires to have a fork (denoted by x and y) after the firing of f in $Proj(C_j)^u$. Thus Case 1.1 is proved.

–Case 1.2. b is reachable in $Proj(C_j)^u$ from s^j (i.e $s^j \xrightarrow{g} ...s1^j \xrightarrow{b}$) but g is blocked in a state s of $||C^u_{1,k}$ by C_1

Then the considerations of Case 1.1 might be repeated for the pair of Sch^u_{seq} states s_b and s_g, where b is fireable in s_b, while g is fireable in s_g.

–Case 2. $C_{2,k}$ is not a concurrent schedule for the subnet $N \setminus C_1$ of the original PN N obtained by deleting from N transitions in C_1, together with their output places.

Then consistency violations are present in the parallel compositions of clusters $C_2, ..., C_k$ and one can repeat the proof for the PN $N \setminus C_1$ and sequential schedule $Sch_{seq} \setminus C_1$ which is obtained from the original Sch_{seq} by projecting on $C_2 \cup ... \cup C_k$.$\diamond$

Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE