Quasi-static scheduling of independent tasks for reactive systems

Jordi Cortadella[1], Alex Kondratyev[2], Luciano Lavagno[3], Claudio Passerone[3], and Yosinori Watanabe[2]

[1] Universitat Politècnica de Catalunya, Barcelona, Spain. jordi@lsi.upc.es
[2] Cadence Berkeley Labs, Berkeley, USA. {kalex,watanabe}@cadence.com
[3] Politecnico di Torino, Italy. {lavagno,claudio.passerone}@polito.it

Abstract. The synthesis of a reactive system generates a set of concurrent tasks coordinated by an operating system. 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. A formal model based on Petri nets is used to synthesize the tasks. A practical application is illustrated by means of a real-life industrial example.

1 Introduction

1.1 Embedded systems

Concurrent specifications, such as dataflow networks [9], Kahn process networks [7], Communicating Sequential Processes [6], synchronous languages [4], and graphical state machines [5], 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 portion of a functional specification allocated to a single processor.

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.

1.2 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 [8] 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 [9]). 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) [1, 10, 12, 2, 13]. 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 [1] for specifications with data-dependent choices . Our work fits in the framework proposed by [12, 2], 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, and thus achieves a two-fold improvement over [1]:

1. undecidability has not been proven nor disproven. It remains an interesting open problem except in the decidable cases of Marked Graphs [9] and Free-Choice Petri nets [12].
2. powerful heuristics, based on the theory discussed in this paper, can be used to speed up the identification of a solution, if it exists.

In the rest of the paper, we define various scheduling problems for Petri nets, and motivate their practical interest by showing how concurrent programs communicating via FIFO queues can be implemented as software tasks running under a Real Time Operating System. In particular, we use a game-theoretic formulation, in which the scheduler must win against an adversary who can determine the outcome of non-deterministic choices, by avoiding overflow of FIFO queues. The scheduler can resolve concurrency in an arbitrary fashion, but is not allowed to "starve" any input by indefinitely refusing to service it.

1.3 Specification model

We consider a system to be specified as a set of concurrent processes, similar to those discussed in [3]. 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.

Each communication action on a port, and each internal computation action is modeled by a *transition* in a corresponding Petri net, while places are used to represent both sequencing within processes and FIFO communication.

Fig. 1 depicts the specification of a concurrent system with two processes, two input ports (`IN` and `COEF`) and one output port (`OUT`). The processes communicate to each other through the channel `DATA`. The process `Get_Data` reads data from the environment and sends it to the channel `DATA`. Moreover, after having sent N samples (N is a constant), it also inserts their average value in the same channel. The process `Filter` extracts the average values inserted by

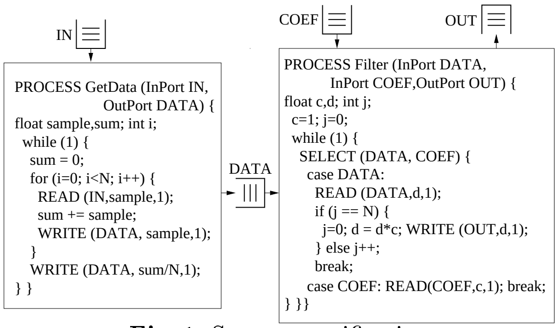


Fig. 1. System specification

`Get_Data`, multiplies them by a coefficient and sends them to the environment through the port `OUT`. This example also illustrates the main extensions of the C language to support communication. The operations to communicate through ports have syntax `READ_DATA (port, data, nitems)` and `WRITE_DATA (port, data, nitems)`. The parameter `nitems` indicates the number of objects involved in the communication. This allows to support multi-rating, although the example uses only 1-object read/write operations. A `READ_DATA` blocks when the number of items in the channel is smaller than `nitems`.

The `SELECT` statement supports synchronization-dependent control, which specifies control depending on the availability of objects on input ports. In the example, the `SELECT` statement in `Filter` non-deterministically selects one of the ports with available objects. In case none of them has available objects, the process blocks until some is available. Figure 2(a) depicts the representation of the concurrent system specified in Fig. 1 with a Petri net model.

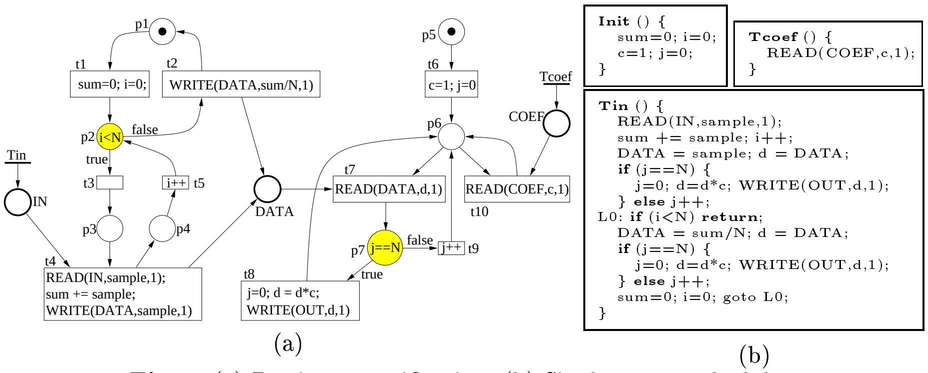


Fig. 2. (a) Petri net specification, (b) Single-source schedules.

1.4 Schedule composition

The main topic of this paper is *schedule composition*. In general a schedule is defined by requiring the system to alternately wait for any one input transition, and perform some internal computation or communication. While this is a very useful theoretical notion, in practice it is much easier to tie inputs of the embedded software to interrupt sources of the processor on which it runs.

Basically, if we were able to define fragments of the schedule so that each of them represented the "maximal amount of work" that could be performed in reaction to each input from the environment, then each such fragment would become an interrupt service routine. When defining such schedule fragments we will mostly focus on their execution in a non-preemptive way, i.e. so that only one of them can be executed at any given time. This results in a simple and general definition of composite schedule, and has the only practical problem that some buffering may be needed on fast inputs in order to process and store the input data while another schedule fragment (called Single Source Schedule in the following) for another input is being executed.

As an example, Fig. 2(b) shows three tasks that implement the behavior of Fig. 1. The task `Init` is executed only at the beginning to reach a steady state. After that, the tasks `Tin` and `Tcoef` are invoked by the arrival of events from `IN` and `COEF`, respectively.

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, $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the flow relation and $M_0 : P \to \mathbb{N}$ 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.*

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.*

Definition 3 (Free-choice set). *A* Free-choice Set *(FCS) is a maximal subset of transitions C such that*

$$\forall t_1, t_2 \in C \text{ s.t. } t_1 \neq t_2 : \; \mathit{Pre}[t_1] = \mathit{Pre}[t_2] \; \wedge \; ({}^\bullet t_1 \neq \emptyset \implies C = ({}^\bullet t_1)^\bullet).$$

Proposition 1. *The set of FCSs is a partition of the set of transitions.*

Proof. The proof immediately follows from the consideration that the relation R induced by FCS (i.e. $t_1 R t_2 \iff \exists \mathsf{FCS}\ C : t_1, t_2 \in C$) is an equivalence relation.

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 particular, T_S is a FCS.

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. A path with multiple transitions can also be denoted by $s \xrightarrow{\sigma} s'$, where σ is the sequence of symbols in the path.

3 Schedules

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.

Definition 5 (Sequential schedule). *Given a Petri net $N = (P, T, F, M_0)$, a* sequential schedule *of N is a transition system $Sch = (S, T, \rightarrow, s_0)$ with the following properties:*

1. *S is finite and there is a mapping $\mu : S \rightarrow [M_0\rangle$, with $\mu(s_0) = M_0$.*
2. *If transition t is fireable in state s, with $s \xrightarrow{t} s'$, then $\mu(s)[t\rangle\mu(s')$ in N.*
3. *If t_1 is fireable in s, then t_2 is fireable in s if and only if $t_2 \in \mathsf{FCS}(t_1)$.*
4. *t is fireable in s_0 if and only if $t \in T_S$.*
5. *For each state $s \in S$, there is a path $s \xrightarrow{\sigma} s' \xrightarrow{t}$ for each $t \in T_S$.*

Property 2 implies a trace containment for Sch and N (any feasible trace in the schedule is feasible in the original PN). Property 3 indicates that one FCS is scheduled at each state. The FCS scheduled at s_0 is the set of source transitions (property 4). Finally, property 5 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 only source transitions 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 (property 4 of definition 5).
– The scheduler might pick up any of the enabled transitions to fire (property 3) 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 places).

  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.

The goal of the game is to process any input from the environment (property 5) 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 arbitration with two processes competing for the same resource (place p_0). The schedule for this specification is given in Fig. 4(b), where await states are shown by shadowed rectangles. Note that the scheduler makes a smart choice on which one among the concurrently enabled transitions a, d or f fires in the state $\{p_4, p_5\}$, by first scheduling transition f to release the common resource p_0 as quickly as possible.

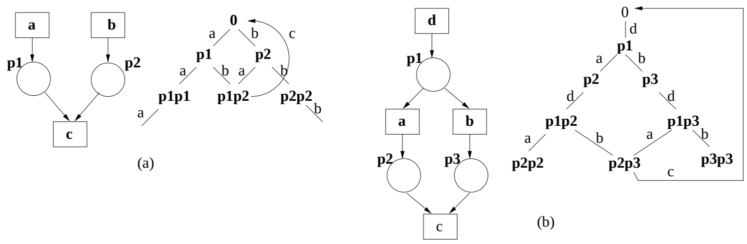


Fig. 3. Non-schedulable PNs

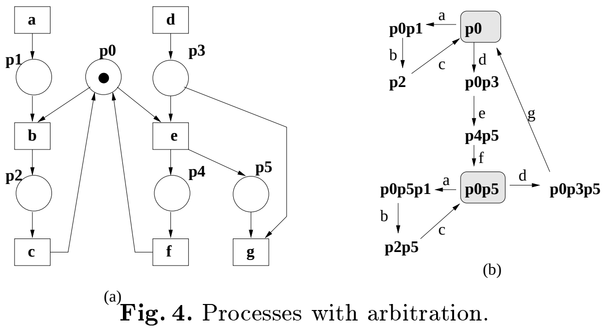


Fig. 4. Processes with arbitration.

3.1 Single-source schedules: rationale

As discussed in Section 1, the proposed strategy synthesizes a set of tasks that *serve* the inputs events produced by the environment and may share common data structures in the system. Therefore, their interaction must be consistent, independent of the occurrence order of the external events.

The concept of task in the reactive system corresponds to the concept of single-source schedule (SSS) in our formal model. A SSS is a sequential schedule associated to a single source transition. Each SSS serves only one input channel as if other source transitions were never produced by the environment. In that way a SSS gives a projection of the scheduler activity in which only one source transitions is fireable.

Given a set of SSSs, we want to check whether it can implement the specification of the system. For that, we need to calculate their composition and check that it fulfills the properties of *sequential schedule* (see definition 5).

The rationale behind generation of SSSs in first place rather than constructing a sequential schedule is the following:

- Lower complexity for the generation of SSSs. The size of a sequential schedule can be exponentially larger than the size of the set of SSSs.
- SSSs give a natural decomposition of a sequential schedule which is beneficial for implementation as ISRs on an RTOS.
- A scheduler that behaves according to SSSs has a uniform response for firings of the same source transitions, since each SSS often has just a single await state. This uniformity can be exploited during code generation and provides potentially smaller code size due to the higher probability for sharing pieces of code.

3.2 Single-source schedules: definition and composition

Definition 6 (Single-source schedule). *Given a Petri net* $N = (P, T, F, M_0)$*, a* single-source schedule *of* N *with the transition* $a \in T_S$ *is a sequential schedule of* $N \setminus (T_S \setminus \{a\})$

Next, sequential composition is defined. The intuitive idea behind this composition is as follows. Each transition system represents a task associated to a source transition. When a task is active, it cannot be preempted, i.e. only events from that task can be issued. A task can only be preempted when it is waiting for an event from the environment (source transition). The composition builds a system that can serve all the events of the environment sequentially.

Definition 7 (Sequential composition). *Let $N = (P, T, F, M_0)$ be a Petri net and $\mathcal{X} = \{\mathsf{SSS}(t_i) = (S_i, T_{t_i}, \rightarrow_i, s_{0_i}) \mid t_i \in T_S\}$ be a set of SSSs of N. The sequential composition of $\mathcal{X}$ is a transition system $A = (S, T, \rightarrow, s_0)$ defined as follows:*

- $s_0 = (s_{0_1}, \ldots, s_{0_k})$
- *$S \subseteq S_1 \times \cdots \times S_k$ is the set of states reachable from s_0 according to $\rightarrow$. A state is called an* await *state if all its components are await states in their corresponding* SSS.
- *For every state $s = (s_1, \ldots, s_k)$,*
  - *if s is an await state, then the set of fireable transitions from s is the set of source transitions, i.e. $(s_1, \ldots, s_i, \ldots, s_k) \xrightarrow{t_i} (s_1, \ldots, s_i', \ldots, s_k)$ in A if and only if $s_i \xrightarrow{t_i} s_i'$ in $\mathsf{SSS}(t_i)$.*
  - *if s is not an await state, there is one and only one*[1] *state component s_i of s such that s_i is not an await state in $\mathsf{SSS}(t_i)$. Then the set of fireable transitions from s is the set of fireable transitions from s_i in $\mathsf{SSS}(t_i)$, i.e. $(s_1, \ldots, s_i, \ldots, s_k) \xrightarrow{t} (s_1, \ldots, s_i', \ldots, s_k)$ in A if and only if $s_i \xrightarrow{t} s_i'$ in $\mathsf{SSS}(t_i)$.*

Figure 5 depicts the sequential composition of two SSSs obtained from the PN in Fig. 4. The shadowed circles correspond to await states. Initially both SSSs are in await states. Thus, only source transitions a and d are fireable in state 00 of the composition. The firing of any of them, e.g. d, moves the corresponding SSS from the await state and forces the composition to proceed according to the chosen SSS(d) until a new await state is reached (state 3 of SSS(d)). In the corresponding state of the composition (state 03) both state components are await states and, therefore, both source transitions a and d are fireable again.

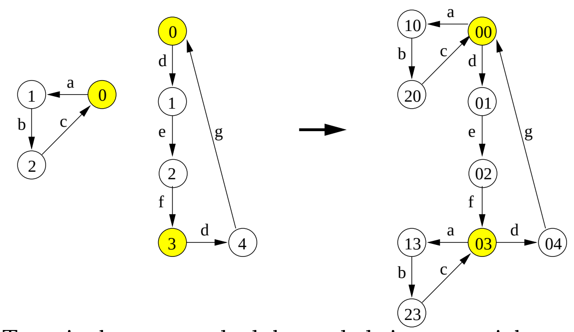


Fig. 5. Two single-source schedules and their sequential composition.

[1] This claim can be easily proved by induction from the definition of $\rightarrow$ and from the fact that s_0 is an await state.

Definition 8 (Sequential independence). *Given a Petri net* $N = (P, T, F, M_0)$*, a set of single-source schedules* $\mathcal{X}$ *is* sequentially independent *if its sequential composition is isomorphic to a sequential schedule of* N*.*

One can easily check that the sequential composition in Fig. 5 is isomorphic to the sequential schedule in Fig. 4(b) and, therefore, the set $\{\mathsf{SSS}(a), \mathsf{SSS}(b)\}$, is sequentially independent.

From the definition of SSS, it follows that the existence of a sequential schedule implies the existence of SSSs (once a sequential schedule has been obtained all SSSs can be immediately derived by using the sub-graphs in which only one source transition fires). Moreover, Definition 8 indicates that sequential independence for a set of SSSs is a sufficient condition for the existence of a sequential schedule. In fact, it even gives a constructive way for deriving such a schedule by using the sequential composition of SSSs. For this reason, checking the independence of a set of SSSs is a key issue in the suggested approach.

3.3 Checking sequential independence

Given a Petri net N and a set $\mathcal{X}$ of single-source schedules of N, checking their independence can be done as follows:

1. Build the sequential composition A of $\mathcal{X}$.
2. Check that A is a sequential schedule of N, according to Definition 5.

This approach is computationally expensive because it requires to derive explicitly the composition of SSSs.

We next propose an alternative way for checking independence of SSSs that does not require to calculate their composition. Let us consider the case in which the SSSs are not independent, resulting in a failure to find an isomorphic sequential schedule Sch for A. Let us consider paths from the initial states of A and Sch, where Sch mimics A and keeps track of the reachable markings in the Petri net. If there is no independence, there will be two paths that lead to states s and s' in A and Sch, respectively, in which some transition t is enabled in s but not enabled in s', i.e. the Petri net cannot simulate the sequential composition of SSSs. Figure 6 shows the structure of the paths, where shadowed circles denote await states.

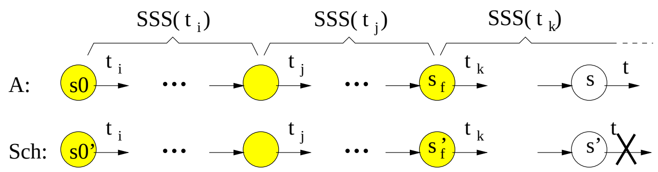


Fig. 6. Matching SSS composition with a sequential schedule.

In the last await state s_f before s, $\mathsf{SSS}(t_k)$ is chosen to proceed in the composition by firing transition t_k. The only reason for t not being enabled in state $s' \in Sch$ might come from the "interference" of the execution of the schedules

$\mathsf{SSS}(t_i)$ and $\mathsf{SSS}(t_j)$ preceding s_f with $\mathsf{SSS}(t_k)$. Simply speaking, $\mathsf{SSS}(t_i)$ and $\mathsf{SSS}(t_j)$ must consume tokens from some place p in the pre-set of t. This leads to the idea of applying marking equations for the check of SSS independence. It is known that self-loops introduce inaccuracy in calculating the fireable transitions by using the marking equations. For the rest of this section we will assume that a specification is provided as a PN without self-loops [2]. The following hierarchy of notions is used for the formulation of independence via marking properties:

- For $\mathcal{X} = \{\mathsf{SSS}(t_i) \mid t_i \in T_S\}$ and given place p
  - For $\mathsf{SSS}(t_i)$ with set of states S_{t_i} and set of await states $S^a_{t_i}$
    - For state $s \in S_{t_i}$ let $change(p, s) = \mu(s_0)(p) - \mu(s)(p)$, i.e. the difference in token counts for place p between markings corresponding to initial state of $\mathsf{SSS}(t_i)$ and state s.
    - let $\mathsf{SSS}\_change(p, t_i) = \max_{s \in S_{t_i}} change(p, s)$, i.e. the maximal change in token count for place p in markings corresponding to states of $\mathsf{SSS}(t_i)$ with respect to the initial marking.
    - let $await\_change(p, t_i) = \max_{s \in S^a_{t_i}} change(p, s)$, i.e. the maximal change in token count for place p in markings corresponding to await states of $\mathsf{SSS}(t_i)$ with respect to the initial marking.
  - let $worst\_change(p, t_i) = \sum_{t_j \neq t_i, t_j \in T_S} await\_change(p, t_j)$, i.e. the sum of $await\_change$ for all SSS except for $\mathsf{SSS}(t_i)$

Here is the semantics of the introduced notions:

- $\mathsf{SSS}\_change(p, t_i)$ shows how much the original token count for place p deviates while executing the single source schedule $\mathsf{SSS}(t_i)$. If $\mathsf{SSS}(t_i)$ started from the initial marking with a number of tokens in p less than $\mathsf{SSS}\_change(p, t_i)$ then $\mathsf{SSS}(t_i)$ would deadlock due to a lack of tokens to fire some transition in the post-set of p.
- $await\_change(p, t_i)$ gives a quantitative measure of the influence of $\mathsf{SSS}(t_i)$ on the other schedules. Indeed, as await states are the only points where a scheduler switches among interrupt service routines ($\mathsf{SSS}s$), the change in PN markings due to the execution of $\mathsf{SSS}(t_i)$ is fully captured by the markings of await states, where $await\_change(p, t_i)$ gives the worst possible scenario.
- $worst\_change(p, t_i)$ generalizes the notion of $await\_change(p, t_i)$ to the set of all SSSs except for the chosen $\mathsf{SSS}(t_i)$. The execution of other SSSs has a cumulative influence on $\mathsf{SSS}(t_i)$ expressed by $worst\_change(p, t_i)$.

The following theorem establishes a bridge between the sequential independence of SSS and the firing rules in Petri nets when the schedules are executed.

Theorem 1. *A set of single source schedules $\mathcal{X} = \{SSS(t_i) \mid t_i \in T_S\}$ derived from a self-loop free PN $N = (P, T, F, M_0)$ is sequentially independent if and only if $\forall p \in P$ and $\forall SSS(t_i) \in \mathcal{X}$ the following inequality is true:*

$$M_0(p) - worst\_change(p, t_i) - SSS\_change(p, t_i) \geq 0 \qquad \text{(IE.1)}$$

[2] This requirement does not impose restrictions because any PN with self-loops can be transformed into a self-loop-free PN by inserting dummy transitions.

Proof. $\Rightarrow$. Suppose that $\mathcal{X}$ is sequentially independent but there exists a place p for which inequality IE.1 is not satisfied. Sequential independence implies the existence of a sequential schedule isomorphic to the composition of $\mathcal{X}$.

In the set of states of the sequential composition of $\mathcal{X}$ let us choose an await state $s = (s_1, \ldots, s_k)$, such that for any $\mathsf{SSS}(t_j), t_j \neq t_i$ the corresponding await component s_j of s is chosen to maximize the token consumption in place p, while s_i is chosen to be the initial state of $\mathsf{SSS}(t_i)$. From the choice of state s follows that by reaching s in the composition, the corresponding marking for place p equals to $M_0(p) - worst\_change(p, t_i)$. Let us execute $\mathsf{SSS}(t_i)$ from s. By the definition of $\mathsf{SSS}\_change(p, t_i)$ there is a state $s_i' \in \mathsf{SSS}(t_i)$ such that the token count for place p in the marking corresponding to s_i' reduces by $\mathsf{SSS}\_change(p, t_i)$ with respect to the initial marking from which $\mathsf{SSS}(t_i)$ starts. From this follows that if $M_0(p) - worst\_change(p, t_i) - \mathsf{SSS}\_change(p, t_i) < 0$ then in the sequential schedule isomorphic to the sequential composition of $\mathcal{X}$ it would be impossible to fire some transition t that enters state s', where $s' = (s_1, \ldots, s_i', \ldots, s_k)$. The latter contradicts the isomorphism between the composition and the sequential schedule

$\Leftarrow$. Suppose that inequality IE.1 is satisfied but $\mathcal{X}$ is not sequentially independent. In a set of all sequential schedules let us choose the schedule Sch that is isomorphic to the largest subpart of the sequential composition A, i.e. if a mismatch like in Fig. 6 is found by simulating Sch and A then there does not exist any other sequential schedule with state s'' isomorphic to s and capable of firing transition t. Let us rearrange the sequence in Fig. 6 by first executing the schedules other than $\mathsf{SSS}(t_i)$ and let s_f be the first await node in which $\mathsf{SSS}(t_i)$ is chosen. Then the token count for a place p in the marking corresponding to s_f is larger than $M_0(p) - worst\_change(p, t_i)$. By definition the execution of $\mathsf{SSS}(t_i)$ cannot reduce it more than by $\mathsf{SSS}\_change(p, t_i)$. Then due to the validity of IE.1 when state s' is reached in $\mathsf{SSS}(t_i)$, transition t cannot lack tokens in p needed for its enabling. The case of Fig. 6 is impossible.

A sufficient condition for checking the sequential independence can now be derived.

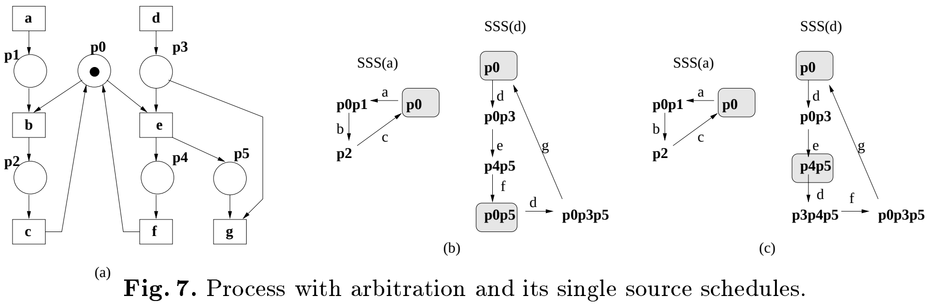


Fig. 7. Process with arbitration and its single source schedules.

Corollary 1. *A set of single source schedules $\mathcal{X} = \{\mathsf{SSS}(t_i)\}$ is sequentially independent if for any marking M corresponding to an await state s of $\mathsf{SSS}(t_i)$ ($M = \mu(s)$) we have $\forall p : M(p) \geq M_0(p)$.*

Proof. The proof follows from inequality IE.1 by taking into account two observations:

– If for any marking M of the await state s $M(p) \geq M_0(p)$, then $worst\_change(p, t_i) \leq 0$.
– The ability of any $\mathsf{SSS}(t_i)$ to be executed from M_0 means that for any place p, $M_0(p) - \mathsf{SSS}\_change(p, t_i) \geq 0$. Note that this captures the case of arbitrary PNs (not self-loop free only).

We illustrate the suggested approach with the example in Fig. 7. The two different sets of SSSs are shown in Fig. 7(b,c). The only place shared by both $\mathsf{SSS}(a)$ and $\mathsf{SSS}(d)$ is the place p_0. We can immediately infer the irrelevance of other places with respect to independence violation. Checking the marking count for p_0 in $\mathsf{SSS}(d)$ in Fig. 7(b) gives the following results: $worst\_change(p_0, d) = 0$ (p_0 is marked in both await nodes of $\mathsf{SSS}(d)$) and $\mathsf{SSS}\_change(p_0, d) = 1$ due to the consumption of p_0 in non-await states of $\mathsf{SSS}(d)$ (see the marking $\{p_4, p_5\}$ e.g.). From similar considerations: $worst\_change(p_0, a) = 0$ and $\mathsf{SSS}\_change(p_0, a) = 1$. It is easy to see that under the initial marking $M_0(p_0) = 1$ inequality IE.1 is satisfied for both $\mathsf{SSS}(a)$ and $\mathsf{SSS}(d)$. This is in full correspondence with the conclusion about the sequential independence of $\mathsf{SSS}(a)$ and $\mathsf{SSS}(d)$ that was derived earlier through the explicit construction of their composition (see Fig. 5).

Reversing the order of firing for transitions d and f in $\mathsf{SSS}(d)$ from Fig. 7(c) results in $worst\_change(p_0, d)$ increasing to 1 (in the await state $\{p_4, p_5\}$ place p_0 is unmarked). The latter leads to the violation of inequality IE.1 for $\mathsf{SSS}(a)$ and tells about the dependency between $\mathsf{SSS}(a)$ and $\mathsf{SSS}(d)$ from Fig. 7(c). Note that the same result could be immediately concluded by observing await states of $\mathsf{SSS}(d)$ and applying Corollary 1.

From the above example it follows that from the same specification one can obtain independent and dependent sets of SSSs. In case an independent set exists, finding it can be computationally expensive, since an exhaustive exploration of the concurrency in all SSSs may be required. In practice, we suggest to use a "try and check" approach in which a set of SSSs is derived and, if not independent, a sequential schedule is immediately constructed (if possible). This design flow for scheduling is illustrated by Fig. 8.

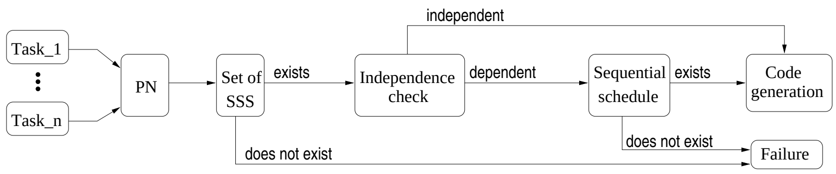


Fig. 8. Design flow for quasi-static scheduling.

3.4 Termination criteria

Single-source schedules are derived by exploring the reachability graph of a Petri net with source transitions. Unfortunately, this graph is infinite.

Next, we discuss conservative heuristic approaches to prune the exploration of the reachability space while constructing a schedule. Conservatism refers to the fact that schedules may not be found in cases in which they exist. Our approach attempts to prune the state space when the search is done towards directions that are qualified as *non-promising*, i.e. the chances to find a valid schedule are remote. The approach is based on defining the notion of *irrelevant marking*. This definition is done in two steps: 1) bounds on places are calculated from the structure of the Petri net and 2) markings are qualified as irrelevant during the exploration of the state space if they cover some preceding marking and exceed the calculated bounds. Note that the property of irrelevance is not local and depends on the pre-history of the marking.

Definition 9 (Place degree). *The degree of a place p is defined as:*

$$degree(p) = \max(M_0(p), \max_{t \in {}^\bullet p} F(t,p) + \max_{t \in p^\bullet} F(p,t) - 1)$$

Place degree intuitively models the "saturation" of p. If the token count of p is $\max F(p,t)$ or more, then adding tokens to p cannot help in enabling output transitions of p. By the firing of a single input transition of p at most $\max F(t,p)$ tokens can arrive, which gives the expression for place degree shown in Definition 9.

Definition 10 (Irrelevant marking). *A marking M is called* irrelevant *with respect to a reachability tree rooted in initial marking M_0, if the tree contains marking M_1 such that:*

- *M is reachable from M_1,*
- *no place has more tokens in M_1 than in M, and*
- *for every place p at which M has more tokens than M_1, the number of tokens in M_1 is equal to or greater than $degree(p)$.*

The example in Fig. 9 illustrates the crucial difference between the approaches targeted to pre-defined place bounds and irrelevant markings.

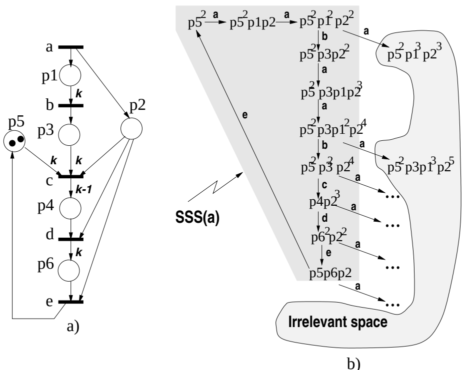


Fig. 9. Constraining the search space by irrelevance criterion

The maximal place degree in PN from Fig. 9(a) is k. This information is the best (as far as we know) one can extract from the PN structure about place bounds. The predefined upper bounds for places should be chosen to exceed place degrees. In fact, the higher place degrees are, the higher upper bounds are expected. Suppose that based on this rationale the upper bounds are chosen as maximal place degree multiplied by some constant margin.

Let us assume for our example that place bounds are assigned to be $2k-1$ and consider the PN reachability space when $k = 2$. When the schedule is checked with the pruning based on pre-defined place bounds, any marking that has more than 3 tokens in a place should be discarded. Clearly no schedule could be found in that reachability space because after a, a, b, a occurs, the only enabled transition is a, but its firing produces 4 tokens in place $p2$ (see the part of reachability graph shown in Fig. 9(b), where superscripts near places show the number of tokens the place has under the current marking). The search fails.

The irrelevance criterion handles this problem more graciously. It guides the search for the "proper" direction in the reachability space by avoiding the irrelevant markings. The first guidance is given when marking $\{p_5^2, p_1^2, p_2^2\}$ is reached. In that marking one need to choose which transitions a or b to fire from the enabled set. The firing of a however produces the marking $\{p_5^2, p_1^3, p_2^3\}$ which is irrelevant because it covers $\{p_5^2, p_1^2, p_2^2\}$, where places p_1 and p_2 are already saturated. Therefore transition b should be chosen to fire. After this, a fires two times, resulting in the marking $\{p_5^2, p_3, p_1^2, p_2^4\}$. Note that even though the place degree for p_2 is exceeded in this marking, the marking is not irrelevant because in all the preceding markings containing p_3, p_1 is not saturated. From this marking the system is guided to fire b because the firing of a again would enter the irrelevant space (see Fig. 9(b)). Finally this procedure succeeds and finds a valid SS schedule.

Though pruning the search by using irrelevance seems a more justified criterion than by using place bounds, it is not exact for general PNs. There exist PNs for which any possible schedule enters the irrelevant space. This is due to the fact that for general PNs accumulating tokens in choice places after their saturation could influence the resolution of choice (e.g., by splitting token flows in two choice branches simultaneously). If for any choice place p in PN either at most one of transitions in $p^\bullet$ is enabled (unique choice) or every transition in $p^\bullet$ is enabled (free-choice) then adding tokens to p does not change the choice behavior of a PN. This gives the rationale behind the conjecture that *the irrelevant criterion is exact for PNs with choice places that are either unique or free-choice*. However we are unable either to prove the exactness of this criterion or to find a counterexample for that. This issue is open for the moment.

4 Algorithm for schedule generation

This section presents an algorithm for computing a sequential schedule. It can also be used to compute a single-source schedule for a source transition t_i, if it takes as input the net in which all the source transitions except t_i are deleted (see Definition 6). Finally, a sequential program is generated from the resulting schedule by the procedure described in Sect. 4.2.

4.1 Synthesis of sequential schedules

Given a PN N, the scheduling algorithm creates a directed tree, where nodes and edges are associated with markings and transitions of N respectively. In the sequel, $\mu(v)$ denotes the marking associated with a node v of the tree, while $T([v, w])$ denotes the transition for an edge $[v, w]$. Initially, the root r is created and $\mu(r)$ is set to the initial marking of N. We then call a function EP(r, r), shown in Figure 10(a). If this function returns successfully, the post-processing is invoked to create a cycle for each leaf. The resulting graph represents a sequential schedule $(S, T, \rightarrow, r)$, where S is the set of nodes of the graph, T is the set of transitions of N, and $\rightarrow$ is given by $v \overset{T([v,w])}{\rightarrow} w$ for each edge $[v, w]$.

```
function EP(v, target)      // returns (status, ap, ep)
    ap ← 0, ep ← UNDEF, FCS(v) ← φ;
    if(termination conditions hold) return (0, 0,UNDEF);
    if(∃u : u < v and μ(u) = μ(v)) return (1, 0, u);
    for(each FCS F enabled at μ(v))
        if(v = r and F ≠ T_S) continue;      // r is the root.
        if(F = T_S) current_target ← v;
        else current_target ← target;
        (status, ap_F, ep_F) ← EP_FCS(F, v, current_target);
        if(status = 0) continue;
        if(ap_F = 1)
            FCS(v) ← F, return (1, 1, ep_F);
        if(ep_F ≤ current_target)
            FCS(v) ← F, return (1, 0, ep_F);
        if(ep_F < ep)
            FCS(v) ← F, ap ← ap_F, ep ← ep_F;
    if(FCS(v) = φ) return (0, ap, ep);
    else return (1, ap, ep);
```

(a)

```
function EP_FCS(F, v, target)
        // returns (status, ap_F, ep_F)
    ap_F ← 0, ep_F ← UNDEF,
    current_target ← target;
    for(each transition t of F)
        create a node w and an edge [v, w];
        T([v, w]) ← t;
        μ(w) ← the marking obtained by
                    firing t at μ(v);
        (status, ap, ep) ← EP(w, current_target);
        if(status=0) return (0, ap_F, UNDEF);
        if(ap = 1 or FCS(w) = T_S)
            ap_F ← 1, current_target ← v;
        if(ap_F = 0 and v < ep)
            return (0, 0, UNDEF);
        if(ep ≤ v) ep_F ← min(ep_F, ep);
        if(ep_F ≤ target) current_target ← v;
    return (1, ap_F, ep_F);
```

(b)

Fig. 10. The two main functions called in computing a sequential schedule

EP takes as input a leaf v of the current tree and its ancestor *target*. We say that a node u is an *ancestor* of v, denoted by $u \leq v$, if u is on the path from the root to v. If in addition $u \neq v$, u is a *proper ancestor* of v, denoted by $u < v$. EP creates a tree rooted at v, where each node x is associated with at most one FCS enabled at $\mu(x)$. The goal is to find a tree at the root v with two properties. First, each leaf has a proper ancestor with the same marking. Second, each non-leaf x is associated with an FCS so that for each transition t of the FCS, x has a child y with $T([x, y]) = t$. If such a tree is contained in the one created by EP, we say that EP succeeds at v. FCS's are associated so that the conditions given in the definition of sequential schedules (Definition 5) are satisfied, which will be elaborated next.

EP returns three values, denoted by *status(v)*, $ap(v)$, and $ep(v)$. There are two terminal cases, given in the third and fourth lines of the code in Fig. 10(a), for which the returned values are presented respectively. Suppose that v does not fall into the terminal cases. *status(v)* is a boolean variable, which is 1 if and only if EP succeeds at v. The other two values are meaningful only if *status(v)* is 1. $ap(v)$ is a boolean variable, which is 1 if and only if v has a path to an await node in the created tree such that for each edge on the path, say $[x, y]$, an FCS is associated with x and $T([x,y])$ is in the FCS. A node is said to be *await* if it is associated with an FCS and this is the set of source transitions T_S.

$ep(v)$ is an ancestor u of v for which there exists an FCS enabled at $\mu(v)$ that satisfies the following three conditions. First, for each transition t of the FCS, a child w has been created with $T([v,w]) = t$ and $\mu(v)[t\rangle\mu(w)$. Second, for each such w, *status(w)*= 1 and either $ap(w) = 1$ or $ep(w) \leq v$. Third, u is the minimum among $ep(w)$ for all w such that $ep(w) \leq v$, i.e. the one closest to the root r. If no $ep(w)$ is an ancestor of v, or if there is no FCS that satisfies these conditions, $ep(v)$ is set to `UNDEF`. Intuitively, if *ep(v)* is not `UNDEF`, it means that there exists an FCS enabled at $\mu(v)$ with the property that for each transition t of the FCS, if $ep(w) \leq v$ holds for the corresponding child w, there is a sequence of transitions starting from t that can be fired from $\mu(v)$ and the marking obtained after the firing is $\mu(u)$. Further, at each marking obtained during the firing of the sequence, there is an FCS enabled at the marking that satisfies this property. If there exists such an FCS at v, `EP` further checks if there is one that also satisfies $ep(v) \leq target$. If this is the case, `EP` associates one of them with v, which is denoted by $FCS(v)$ in the algorithm. Otherwise, `EP` associates any FCS with the conditions above. If no such FCS exists, no FCS is associated and $FCS(v)$ is set to empty.

To find such an FCS, `EP` calls a function `EP_FCS` for each FCS enabled at $\mu(v)$. The enabled FCS's are sorted in `EP` before calling `EP_FCS`, so that T_S is positioned at the end of the order. This heuristic tries to minimize the number of await nodes introduced in a schedule. The exception of this rule is at the root r, in which `EP_FCS` is called only for T_S. This ensures the property 4 given in the definition of sequential schedules. If `EP` succeeds at the root r, we call the post-processing to create a schedule and terminate. Otherwise, we report no schedule and terminate.

The post-processing consists of two parts. First, we retain only a part of the created tree that are used in the resulting schedule, and delete the rest. The root is retained, and a node w is retained if its parent v is retained and the transition $T([v,w])$ is in $FCS(v)$. Second, a cycle is created for each leaf w of the retained portion of the tree, by merging w with its proper ancestor u such that $\mu(u) = \mu(w)$. By construction, such a u uniquely exists for w. The graph obtained at the end is returned. It is shown that this algorithm always finds a schedule, if there exists one in the space defined by the terminate conditions employed in `EP`.

The resulting graph may have more than one node with the same marking and FCSs associated with these nodes may not be the same. The freedom of associating different FCSs at nodes with the same marking allows the scheduler to explore the larger solution space, and thus the algorithm does not commit to the same FCS for a given marking.

4.2 Code generation

The code generation algorithm takes a graph of a sequential schedule and synthesizes code [11]. We briefly illustrate it in this section.

The algorithm proceeds in three steps. First, we traverse the schedule Sch to identify a set of sub-trees that *covers* Sch, i.e. for each node v of Sch, there exists a tree that contains a node v' with $FCS(v) = FCS(v')$. We say that v *corresponds* to v'. Our procedure finds a minimal set with an additional property that for each v, there is exactly one v' that v corresponds to.

The second step generates code for each tree. This code is made of two types. One is the code for realizing the control flow statements. For example, `if-then-else` is introduced at a node with multiple child nodes. Also, `switch` and `goto` are used to jump from each leaf of the tree to the root of another tree. For this purpose, variables are introduced for places and the markings in the tree are represented with them. This and the correspondence information obtained in the first step are used to implement the jump correctly. The other type of code is operations executed at each transition from one node to another. In our applications, the Petri net is generated so that each transition is annotated with a sequential program. This program is copied in the generated code.

The third step is concerned with channels between processes that have been merged into a single schedule. For each such channel, we define a circular buffer and replace write and read operations for the channel that appear in the generated code with operations on the buffer. The size of the buffer can be statically identified as the upper bound found in the schedule. If the buffer has size 1, it is substituted by a single variable.

5 Experimental Results

We used as our test system an MPEG-2 video decoder developed by Philips (see [14]) and shown in Fig. 11. Processes Thdr and Tvld parse the input video stream; Tisiq and Tidct implement spatial compression decoding; TdecMV, Tpredict and Tadd are responsible for decoding temporal compression and generating the image; Tmemory, TwriteMB, TmemMan and Toutput manage the frame store and produce the output to be sent to a visualization device. Communication is by means of channels that can handle arbitrary data types. Philips used approximately 7700 lines of code to describe the 11 processes, and 51 channels. An average of 16 communication primitives per process are used to transfer data through those channels.

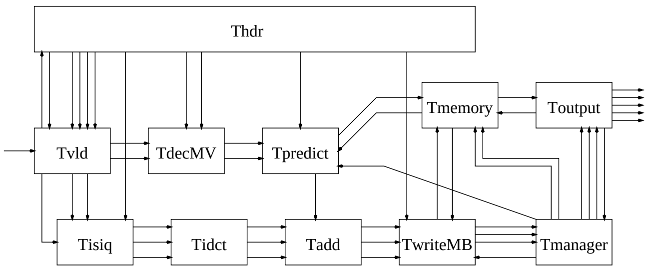
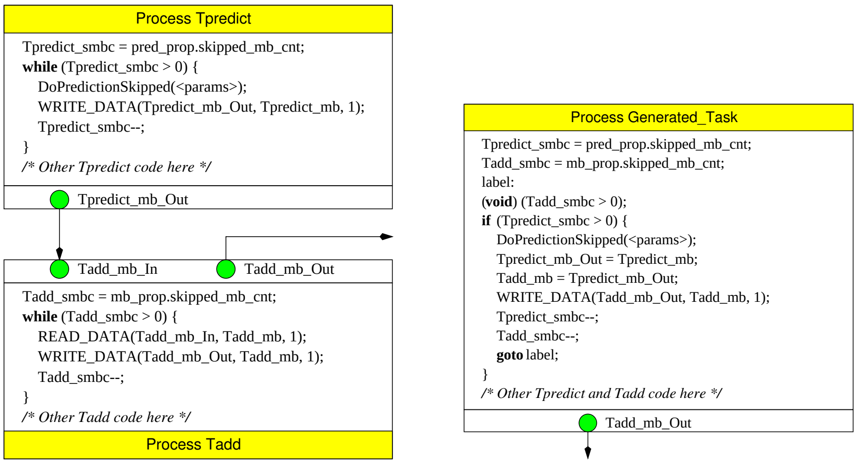


Fig. 11. MPEG-2 video decoder block diagram

In the original implementation, all processes were scheduled at run time. Our objective was to reduce scheduling overhead by merging processes into quasi-statically scheduled ones. We focused our attention on five processes: Tisiq, Tidct, TdecMV, Tpredict and Tadd. They consist of about 3000 lines of code and account for more than half of all communications occurring in the system. The inputs to these five processes from the rest of the system are correlated, and thus they cannot be treated as independent inputs. Instead of modeling the correlation explicitly using additional processes, we introduced a single input that triggers the five processes to react to this input. As a result, our procedure generated a single source schedule for this trigger input.

The Petri net generated from the FlowC specification has 115 places, 106 transitions and 309 arcs. Our algorithm generated a single process with the same interface as the original ones, that could be plugged into the MPEG-2 netlist, replacing the original five processes.

An example of how the code is transformed after scheduling is shown in Fig. 12. Figure 12(a) shows a small fragment of code taken from the processes `Tpredict` and `Tadd`. They both implement a `while` loop during which they exchange some data (a macro-block is written from `Tpredict` to `Tadd`).



(a) (b)

Fig. 12. (a) Example of FlowC specification, (b) Portion of the generated code for the MPEG-2 decoder.

On the other hand, Fig. 12(b) shows the same fragment in the generated process, where the two were merged into a single entity. What is generated is a single loop which contains statements from the two original processes. Note that the `WRITE_DATA` and `READ_DATA` statements in processes `Tpredict` and `Tadd` occurring on the channel connecting them have been transformed into assignments to and from a temporary variable (which can be easily eliminated by an optimizing compiler). The `WRITE_DATA` statement in `Tadd` on the output channel is instead preserved as is, and needs to be expanded to match the communication protocol used in the rest of the system (in our case, it is a FIFO).

We compared the performance of the original specification with that of the same system where a single statically scheduled process. In both cases, we removed the processes that manage and implement the memory, but we kept those that parse the input MPEG stream. Both systems received as input a video stream composed of 4 images (1 intra, 1 predicted, 2 bidirectional predicted).

Table 1 summarizes the total execution time on a Sun Ultra Enterprise 450. It also shows the individual contributions of the processes (split among the parser and the five processes that we scheduled together), the test-bench and the operating system. The increase in performance is around 45%. The gain is concentrated in the statically scheduled processes, due to the reduction in the number of FIFO-based communications, and in the operating system due to the reduction in the number of context switches.

	Total	**MPEG2**			**Test bench**	**OS**
		Total	Parser	5Procs		
Orig.	7.5	4.66	0.94	3.72	0.27	2.58
QSS	4.1	2.51	0.94	1.57	0.28	1.31

Table 1. CPU time, in seconds, of the MPEG-2 example

	5 Processes				
	Total	**Comp.**	**Int. Comm.**	**Ext. Comm.**	**Code size**
Orig.	3.72	1.01	2.23	0.48	18K
QSS	1.57	0.96	0.13	0.48	24K

Table 2. CPU time, in seconds, and code size of the five selected processes

Table 2 compares the execution times due to computation and communication of the five processes, both in the original system and in the quasi-statically scheduled one. As expected, computation and external communication are not significantly affected by our procedure. However, internal communication is largely improved: this is because after scheduling we could statically determine that all channels connecting the five considered processes never have more than one element or structure at a time. Therefore, communication is performed by assignment, rather than by using a FIFO or a circular buffer. The table also report the object code size, which increases in the generated single task with respect to the 5 separated process: this is due to the presence of control structures representing the static schedule in the synthesized code.

6 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. This paper has presented a first effort in automating this bridge. Experiments show promising results and encourage further research in the area.

In the future, we expect a more general definition of the concept of schedule, considering concurrent implementations, and a structural characterization for different classes of Petri nets.

Acknowledgements. This work has been partically funded by a grant from Cadence Design Systems and CICYT TIC 2001-2476.

References

1. J. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model.* PhD thesis, U.C. Berkeley, 1993.
2. 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.
3. 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.
4. N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.
5. 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.
6. 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.
7. G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP Congress*, August 1974.
8. H. Kopetz and G. Grunsteidl. TTP – A protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1), January 1994.
9. E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow graphs for digital signal processing. *IEEE Transactions on Computers*, January 1987.
10. B. Lin. Software synthesis of process-based concurrent programs. In *35th ACM/IEEE Design Automation Conference*, June 1998.
11. C. Passerone, Y. Watanabe, and L. Lavagno. Generation of minimal size code for schedule graphs. In *Proceedings of the Design Automation and Test in Europe Conference*, March 2001.
12. 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.
13. 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.
14. P. van der Wolf, P. Lieverse, M. Goel, D.L. Hei, and K. Vissers. An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology. In *Proceedings of the* 7^{th} *International Workshop on Hardware/Software Codesign*, May 1999.