Quasi-Static Scheduling of Independent Tasks for Reactive Systems

Jordi Cortadella, *Member, IEEE*, Alex Kondratyev, *Senior Member, IEEE*, Luciano Lavagno, *Member, IEEE*, Claudio Passerone, *Member, IEEE*, and Yosinori Watanabe, *Member, IEEE*

***Abstract*—A reactive system must process inputs from the environment at the speed and with the delay dictated by the environment. The synthesis of reactive software from a modular concurrent specification model generates a set of concurrent tasks coordinated by an operating system. This paper presents a synthesis approach for reactive software that is aimed 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 and verify the correctness of their composition. A practical application of the approach is illustrated by means of a real-life industrial example, which shows the significant impact of the approach on the performance of the system.**

***Index Terms*—Petri nets, reactive systems, scheduling, software synthesis, specification languages.**

I. Introduction

A. Embedded Systems

THE phenomenal growth of the 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 concurrence.

Concurrent specifications, such as dataflow networks [12], Kahn process networks (KPNs) [11], communicating sequential processes [10], synchronous languages [8], and graphical state machines [9], are interesting because they expose the parallelism inherent in the application, which is much harder to recover *a posteriori* by optimizing compilers. However, their mixed hardware–software implementation on heterogeneous architectures that may include central processing units (CPUs), digital signal processors (DSPs), application-specific integrated circuits (ASICs), coprocessors, field-programmable gate arrays (FPGAs), and so on, requires solving a fundamental *scheduling problem*. We assume in the following that the *allocation* problem of functional processes to architectural resources has already been solved, and we focus on the portion of a functional specification that has been allocated to a single architectural resource, which supports sequential code execution, i.e., a CPU or a DSP. In this paper, we address the scheduling problem for that portion, i.e., finding a sequence of operations (a schedule) to be executed for a given concurrent specification that has been allocated to a single resource with sequential code execution. The schedule must have two important properties: 1) it must be able to process any input from the environment in a finite amount of time; and 2) it must use a finite amount of resources (capacity of buffers).

Although an extension of the suggested approach to concurrent implementation architectures is possible (see, e.g., [6]), this topic is outside the scope of the paper.

B. Static and Quasi-Static Scheduling

Static scheduling techniques solve the problem at compile time. The resulting behavior is thus highly predictable and the overhead due to task context switching is reduced. They may also achieve very high resource utilization if the arrival rate of inputs from the environment is regular and predictable at compile time. Static scheduling, however, is limited to specifications without run-time choice, and researchers have 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) [3]–[5], [13], [18], [19], [22]. It generates one or more tasks that can then be managed by a (possibly preemptive, as we will see in Section III-B) real-time operating system (RTOS).

The QSS problem has been proven to be undecidable by [3] for specifications with data-dependent choices in dataflow networks under the requirement of finite buffers. Thus, any proposed solution is necessarily heuristics, which either bounds the capacity of buffers *a priori* [13], [19], [20], or provides only sufficient conditions and may fail to schedule specifications that indeed have valid schedules [3], [5], [18]. We show in Section III-E that such bounds are very difficult to derive *a priori* for some legitimate networks, and provide an alternative solution based on heuristics that we conjecture to be exact for a class of specifications.

C. Specification Model

Our work fits in the framework proposed by [5] and [18], in that Petri nets (PNs) are used as an abstract model of the specification for which schedules are sought.

We consider a *system* to be scheduled as a network of communicating sequential processes. In this work, we use an

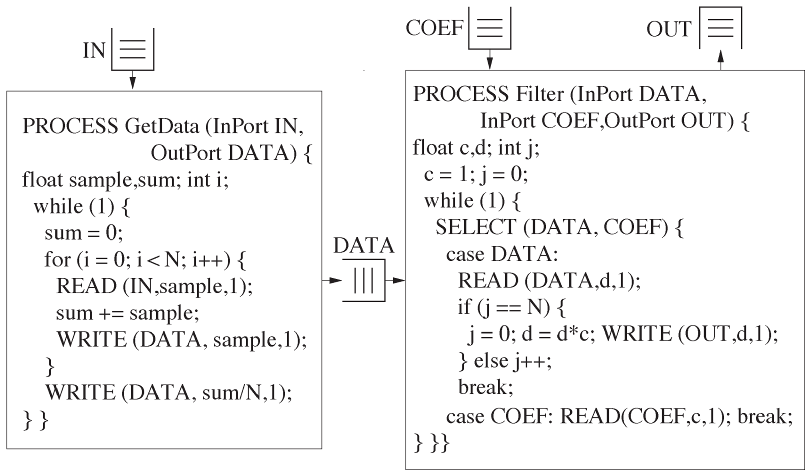
Manuscript received March 30, 2004. This work was supported in part by a grant from Cadence Design Systems and in part by CICYT TIN2004-07925. This paper was recommended by Associate Editor R. Camposano.

J. Cortadella is with the Universitat Politècnica de Catalunya, 08034 Barcelona, Spain (e-mail: jordi.cortadella@upc.edu).

A. Kondratyev and Y. Watanabe are with the Cadence Berkeley Laboratories, San Jose, CA 95134 USA (e-mail: kalex@cadence.com; watanabe@cadence.com).

L. Lavagno and C. Passerone are with the Politecnico di Torino, 10129 Turin, Italy (e-mail: lavagno@polito.it; passerone@polito.it).

Digital Object Identifier 10.1109/TCAD.2005.852038



```
PROCESS GetData (InPort IN,
                 OutPort DATA) {
float sample,sum; int i;
  while (1) {
    sum = 0;
    for (i = 0; i < N; i++) {
      READ (IN,sample,1);
      sum += sample;
      WRITE (DATA, sample,1);
    }
    WRITE (DATA, sum/N,1);
} }
```

```
PROCESS Filter (InPort DATA,
        InPort COEF,OutPort OUT) {
float c,d; int j;
  c = 1; j = 0;
  while (1) {
    SELECT (DATA, COEF) {
      case DATA:
        READ (DATA,d,1);
        if (j == N) {
          j = 0; d = d*c; WRITE (OUT,d,1);
        } else j++;
        break;
      case COEF: READ(COEF,c,1); break;
} }}
```

Fig. 1. System specification.

extension of basic KPNs [11] called Y–chart applications programmers interface (YAPI) [7], which adds a nondeterministic SELECT mechanism to increase the efficiency of handling noncorrelated input streams. A set of input and output ports are defined for each process, and point-to-point communication between processes occurs through unidirectional first in, first out (FIFO) queues between ports. These queues are referred to as *channels*. Multirate communication is supported, i.e., the number of data objects read or written by a process at any given time may be an arbitrary constant.

The control flow of the system is represented by a PN. Each communication action on a port, and each internal computation action, is modeled by a *transition*. 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).

In our synthesis framework, the functionality of the system is described in a C-based language called FlowC, which extends C with interprocess 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 with each other through the channel DATA.

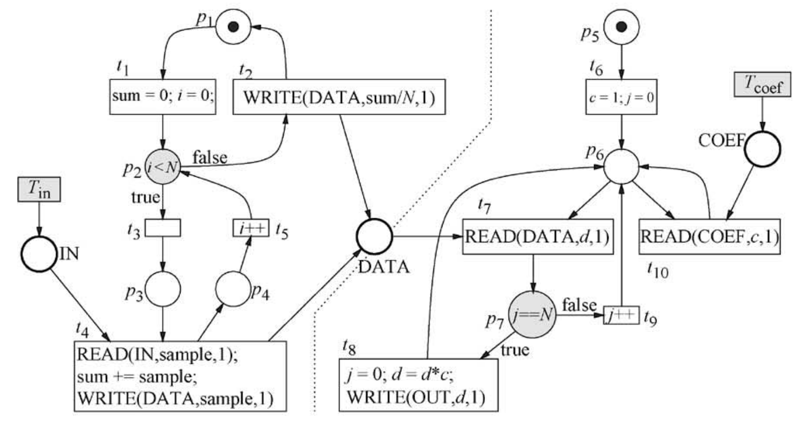
The process GetData reads data from the environment and sends it to the channel DATA. Moreover, after having sent N samples (N is a constant), it also inserts their average value in the same channel. The process Filter extracts the average values inserted by GetData, multiplies them by a coefficient, and sends them to the environment through the port OUT.

The operations to communicate through ports have syntax READ(port, data, nitems), and WRITE(port, data, nitems). The parameter nitems (a compile-time constant, in order to be translated into a PN arc weight) indicates the number of data objects involved in the communication. This permits the support of multirating, although the example uses only one-object read/write operations. A READ 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 data objects on input ports. In the example, the SELECT statement in Filter nondeterministically selects one of the ports with available data objects. In case none of them has available data objects, the process blocks until some data becomes available. SELECT is a crucial statement to model reactive systems with several input ports, where the system is often waiting for the occurrence of events at any of the ports and reacts by nondeterministically choosing one of them. As mentioned above, this is the key difference between KPNs and YAPI, and it allows modeling, e.g., a user-command stream arriving at a nondeterministic rate unrelated to that of a video-frame stream, without requiring the (computationally expensive) trick of defining empty tokens to mark the absence of data objects.

Fig. 2(a) depicts the representation of the concurrent system specified in Fig. 1 with a PN model. This model is the one used to synthesize a schedule.

Note that data-dependent control is modeled as nondeterminism in the PN. This means that we may pessimistically reject some schedulable specification as nonschedulable. For example, two data-dependent control constructs in a specification may be correlated, so that if one of them is resolved in a particular way, then the other is always resolved uniquely. Even in such case, we model each construct independently as a nondeterministic choice, and thus scheduling algorithms built for this model fail to account for the correlation. We deliberately decided to use this modeling mechanism despite such potential drawbacks, for two reasons. First, this simple, structural representation of data-dependent controls allows us to employ efficient techniques, based on the structure of the PN, in order to analyze the system behavior, which often leads to finding a valid schedule very quickly in practice. The techniques are presented in Section IV and their effectiveness is shown by the experimental results. Second, correlated control constructs are often identified before the scheduling phase in practical design flows, either by simulation of the application code or directly from the specification documents. Our tool environment allows the users to specify such correlation through a simple user interface on top of the FlowC code, and this information is automatically translated to a PN structure that is added to the original PN obtained from the FlowC code. In this way, the correlation is represented in the structure of the resulting PN, and thus, the



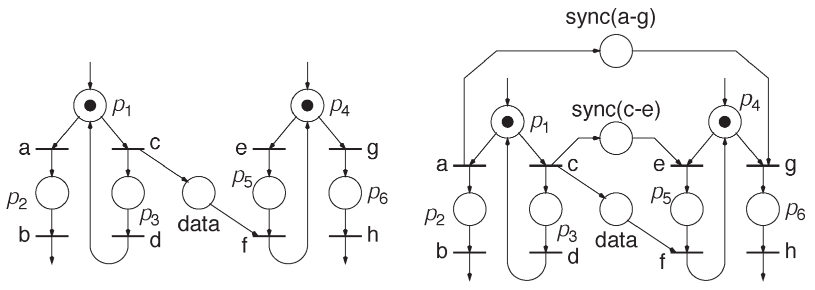
```
Init ()
    sum = 0; i = 0;
    c = 1; j = 0;

Tcoef ()
    READ(COEF,c,1);

Tin ()
    READ(IN,sample,1);
    sum += sample; i++;
    DATA = sample; d = DATA;
    if (j == N)
        j = 0; d = d*c; WRITE(OUT,d,1);
        else j++;
L0: if (i < N) return;
    DATA = sum/N; d = DATA;
    if (j == N)
        j = 0; d = d*c; WRITE(OUT,d,1);
        else j++;
    sum = 0; i = 0; goto L0;
```

(a) (b)

Fig. 2. (a) PN specification. (b) Single-source schedules.



(a) (b)

Fig. 3. Correlated choices. (a) Original PN. (b) Translated PN.

same structural techniques can be used for finding schedules, but this time with the specified correlation taken into account.

An example of such translation is shown in Fig. 3: let us assume that choices represented by places p_1 and p_4 in Fig. 3(a) are correlated, such that whenever transition c is chosen, transition e should be chosen, and whenever transition a is chosen, transition g should be chosen. However, since the two choices are structurally independent, our scheduling algorithm would explore other combinations as well. On the other hand, if the correlation is specified by the designer, the resulting PN would be the one shown in Fig. 3(b), which contains two additional places that model the correlation. This restricts the reachability space of the system and guides the scheduling algorithm in exploring relevant combinations of choice outcomes, avoiding those that are not possible in reality. We have found that this mechanism works effectively in practice [1], and that structural nondeterminism is an effective model for data-dependent controls, in the context of the theoretically undecidable scheduling problem that we address in this paper.

In formulating the scheduling problem precisely, we need to clarify the model and assumptions employed for representing the behavior of the environment. We model the inputs from the environment using *source* transitions, as depicted by T_{in} and T_{coef} in Fig. 2(a). We consider two types of inputs, and distinguish them by associating with each source transition, the type of the input modeled by it. The types are called *controllable* and *uncontrollable*, respectively. The uncontrollable inputs are the stimuli to the system being scheduled, i.e., the system execution takes place as a reaction to events provided by this type of inputs. The objective of the scheduling algorithm is thus to find a finite sequence of operations to be executed in each such a reaction. We formulate the scheduling problem under the assumption that all the uncontrollable inputs are independent with respect to each other, and with respect to the execution of the system. This means that the system cannot tell when the stimuli are provided by the environment or how they are related, and thus, no such information can be assumed when schedules are sought. Therefore, a schedule must be designed so that when the system is ready to react to a stimulus from one uncontrollable input, it must be ready to react to a stimulus from any other uncontrollable input. Consumption of such stimuli by the system is specified in FlowC by the READ primitive introduced earlier. In Fig. 1, all the inputs are uncontrollable.

Controllable inputs, on the other hand, represent data from the environment that the system can acquire whenever it decides to do so. It follows that schedules can be sought under the assumption that if the read operation is executed on a controllable input, then the operation will always succeed in reading the specified amount of data from the input without blocking the execution of the system. In this sense, there is no semantical difference in the context of the scheduling problem between read operations on controllable inputs and internal operations that do not access ports. As with uncontrollable inputs, the read operations are specified with the READ operator in FlowC.[1]

The assumption of mutual independence among the uncontrollable inputs does not prevent us from addressing the case where they are, in fact, dependent. In practice, our limited classification of the types of inputs may force one to categorize inputs as uncontrollable, if they would be treated as stimuli to trigger reactions from the system, even though their firing rates have some correlation. In this case, a valid schedule under our assumption is still a valid schedule of the system; it only implies that the execution of a part of the schedule does not occur because of the correlation of the uncontrollable inputs. Note also that our formulation is conservative, in the sense that we will classify the system as nonschedulable, if we cannot find schedules for parts of the behavior given by our model, although dependence among the uncontrollable inputs may guarantee that the system will never execute any such parts in reality. This situation is similar to that of the use of nondeterministic models for data-dependent controls. That is, even if we treat all such controls independently, we can still handle the case where they are correlated as discussed above, at the expense of potential pessimism, since we may classify some schedulable specifications as nonschedulable, or require user intervention. In fact, one could employ the same technique to structurally represent the correlation between the uncontrollable inputs, in order to make the formulation less conservative.

If the uncontrollable inputs are indeed independent, then a schedule can be defined for each input independently. We refer to such schedules as single-source schedules. If a schedule of the system can be given as an independent set of single-source schedules, then the size of code required for representing the schedule is often much smaller than that for a monolithic schedule which specifies the system execution for all the uncontrollable inputs altogether. In this paper, we first present a formal definition of such monolithic schedules. Single-source schedules are defined as a special subclass of monolithic schedules, and we study some of their properties in Sections III-C and III-D. In particular, we provide a condition under which a set of single-source schedules can be used as a schedule of the system. The scheduling algorithm proposed in Section IV can be applied to either type of schedules, and its experimental results are given in Section V.

[1]Uncontrollable inputs are called "signals" in the reactive language Esterel [8], while controllable inputs are called "sensors."

D. Related Work

Parks addressed the QSS problem in the context of process networks [17] and proposed a procedure that aims to find a schedule with minimal buffer-memory size. The proposed method initially sets a bound on the sizes of the channel buffers, based on the structural properties of the specification, such as the rates given for each read and write primitive, and tries to find a schedule within that bound. If a schedule is not found, then the procedure heuristically increases the sizes of some channels, which causes a deadlock, and repeats the search. In order to claim the absence of a schedule even within a user-given bound for the sizes of the buffers, the reachability space of the system defined for that bound has to be completely analyzed. Since this space can be prohibitively large for practical applications, even for the initial structural bound, the proposed procedure is not effective even for moderate-size examples, for which a schedule does not exist within the initial bound. Further, identifying which buffer sizes should be increased is not straightforward in general, and the effectiveness of the suggested heuristics is not clear. Our algorithm uses techniques based on the PN structure to analyze cyclic behavior obtainable in the reachability space of the system, and searches for schedules without committing to particular bounds (even though it is able to exploit them, if given).

In comparison to the approaches studied based on dataflow networks, such as in [3] and [17], our model has two fundamental differences.

First, these approaches do not have the notion of uncontrollable inputs in the model of the environment. That is, when a process executes read operations on the inputs, it will always succeed in acquiring the needed data, and thus such inputs can be classified as controllable in our model. Therefore, a system with multiple uncontrollable inputs, such as the one shown in Fig. 1, cannot be handled in these approaches without artificial "empty" tokens modeling absence of data.

Second, dataflow networks do not have a mechanism to nondeterministically choose among enabled input ports, given by the SELECT operator in our model. This mechanism is useful, e.g., to efficiently model data-processing filters in which the values of coefficients may be updated nondeterministically, based on dynamic conditions inside the environment. Such applications cannot be handled by dataflow-based approaches, without again requiring empty tokens.

Even though several extensions of conventional dynamic dataflow models were suggested [4], [22] to provide designers with more specification comfort, none of them address the above two issues. We believe that dataflow networks are less convenient than YAPI processes to model complex interactions between multirating and data-dependent choices.

Our approach further differs with respect to other QSS techniques described in [13], [19], and [20], in that our heuristics do not require user-specified bounds in advance. We present a technique that *automatically* identifies bounds on the channel sizes during the schedule search in Section III-E. This technique is known to be a mere heuristics for general PNs, while we conjecture that it is exact for the PNs derived from a FlowC specification without the `SELECT` operator. That is, we conjecture that for this class of specifications, if a schedule does not exist within the bound identified by our technique, then no bounded memory schedule exists for the system.

Our work is complementary to classical real-time scheduling theory (see, e.g., the pioneering work of [14] or, more recently, [2]) in that it maximizes the work done at compile time, and can be viewed essentially as a task-generation approach starting from a functional modular specification. This approach is characterized by the fact that functional units (processes) can be split by this task-generation procedure, if their code belongs to different "computational streams," depending on different, unrelated system inputs.

II. Background

A. PNs and Transition Systems

We assume the reader to be familiar with PNs. The following definitions are presented to introduce the nomenclature used in the paper.

Definition 1 (Petri net): A PN 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 PN is denoted by $[M_0\rangle$. The fact that M' is reachable from M by firing transition t is denoted by $M[t\rangle M'$.

Throughout this paper, we depict a PN using a graph, in which nodes represent the places and transitions, and a directed arc from x to y exists if $F(x, y) \neq 0$.

Definition 2 (Presets and Postsets): Given a node $x \in P \cup T$, the *preset* and *postset* of x are defined as follows:

$$^\bullet x = \{y \mid F(y,x) \neq 0\} \quad x^\bullet = \{y \mid F(x,y) \neq 0\}.$$

Given a PN 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 every node x in N we have $^\bullet x \cap x^\bullet = \emptyset$, then N is called self-loop free. $M(p)$ denotes the number of tokens in place p under marking M. A PN is called *safe* if, for every reachable marking, the number of tokens in any place is not greater than 1.

A transition $t \in T$ is enabled at marking M if its every input place p has a number of tokens greater or equal to $F(p, t)$. An enabled transition t may fire, producing a new marking M' according to the following marking equation: $\forall p : M'(p) = M(p) - F(p, t) + F(t, p)$. A place p that has more than one successor transition is called *choice* place. A pair of transitions t_i and t_j is said to be in a *conflict* if the firing of one of them disables the other.

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

Definition 3 (Source and Nonsource Transitions): The set of transitions of a PN is partitioned into two subsets as follows:

$$T_\mathrm{S} = \{t \in T \mid {}^\bullet t = \emptyset\}, \quad T_\mathrm{N} = T \setminus T_\mathrm{S}.$$

T_S and T_N are the sets of *source* and *nonsource* transitions, respectively. The set of source transitions T_S is further partitioned into *controllable* T_S^c and *uncontrollable* T_S^u ($T_\mathrm{S}^\mathrm{u} = T_\mathrm{S} \setminus T_\mathrm{S}^\mathrm{c}$) 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 III, when we introduce the definition of schedule.

Definition 4 (Free-Choice Set): A *free-choice set* (FCS) is a maximal subset of transitions C, for which one of the two conditions is satisfied: 1) $\forall t_1, t_2 \in C$ s.t. $t_1 \neq t_2$, $^\bullet t_1 \neq \emptyset$: $\mathsf{Pre}[t_1] = \mathsf{Pre}[t_2] \wedge C = (^\bullet t_1)^\bullet$; and 2) $C = T_\mathrm{S}^\mathrm{u}$.

As an example, the sets $\{t_2, t_3\}$ and $\{t_8, t_9\}$ in Fig. 2(a) are FCSs. The set of uncontrollable source transitions $\{T_\mathrm{in}, T_\mathrm{coef}\}$ is also an FCS. However, the set $\{t_7, t_{10}\}$ is not an FCS, since the transitions do not have the same preset.

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

Proof: The proof immediately follows from the consideration of relation R induced by FCS (i.e., $t_1 R t_2 \iff \exists \mathsf{FCS}\ C : t_1, t_2 \in C$). Clearly, R is reflexive, symmetric, and transitive and, therefore, is an equivalence relation. ■

We will call $\mathsf{FCS}(t)$ the set of transitions that belong to the same FCS of t. Informally, an FCS is a set of transitions with the same preset and such that, if it contains more than one transition, none of them has conflicts with any transition outside the FCS. Any conflict inside an FCS is said to be free choice. In particular, T_S^u is an FCS. The enabling of one transition from an FCS implies the enabling of other transitions from the same FCS and the FCS itself is called enabled. If for a choice place p all its successor transitions belong to the same FCS, then p is called free choice. A PN in which all choice places are free choice is called *free-choice* PN.

Definition 5 (Transition System): A transition system (TS) is a 4-tuple $A = (S, \Sigma, \to, s_\mathrm{in})$, where S is a set of states, Σ is an alphabet of symbols, $\to \subseteq S \times \Sigma \times S$ is the transition relation, and s_in is the initial state. Given $s \in S$, $e \in \Sigma$ is said to be *fireable* at s if there exists $s' \in S$ such that $(s, e, s') \in \to$.

With an abuse of notation, we denote by $s \xrightarrow{e} s'$, $s \to s'$, $s \to, \to s, \ldots$ different facts about the existence of a transition with certain properties, where s and s' denote source and destination states, respectively, while $\to$ stands for the transition between them, possibly annotated by the corresponding symbol. For example, $s \xrightarrow{e}$ denotes the fact that there is a transition in $\to$ from state s with symbol e, i.e., the fact that e is fireable in state s.

A path p in a transition system is a sequence of transitions $s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} s_3 \to \cdots \to s_n \xrightarrow{e_n} s_{n+1}$, 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.

Given a transition system $A = (S, \Sigma, \rightarrow, s_{\text{in}})$ a set of states $S' \subseteq S$ defines a *restriction* of A onto S' obtained by removing states $S \setminus S'$ and their adjacent arcs from A. For a given set of states S', one can define subsets of border states by which S' is *entered* $(enter(S') = \{s' \in S' \mid \exists s \notin S' : s \rightarrow s'\})$, and *exited* $(exit(S') = \{s' \in S' \mid \exists s \notin S' : s' \rightarrow s\})$.

B. FlowC-Based PNs

A specification model is represented as a network of communicating processes, each described as a sequential (generally nonterminating) program in *FlowC*. The network of processes is transformed into a single PN, which is built in two steps. In the first step, called *compilation*, a PN for each process is constructed and each port is associated to a place of the PN. The second step, called *linking*, builds a PN by "connecting" the PNs according to the defined channels.

1) Compilation: A specification in *FlowC* is translated into a set of PNs, one for each process, that communicate through ports represented by places. Each transition is annotated with a fragment of C code. Transitions and places are clustered as much as possible in the compilation process, in order to reduce the complexity of the PN, while preserving the structural properties needed for correct scheduling.[2] Processes are sequential and, therefore, their corresponding PNs have no concurrence.

If we ignore the places associated to the ports, the PN of one process obtained by the compilation strategy mentioned above has the following properties: 1) exactly one place is marked at each reachable marking, the token mimics the "program counter" of the sequential process and 2) it is free choice. Choice places represent the evaluation of conditions (e.g., from if–then–else or while statements), and their postset is an FCS.

When places associated to ports are also considered, the underlying PN might no longer be free choice. The possible violations of free choice stem from two sources: 1) when the same process reads data from the same port in different statements, the place representing the port becomes a nonfree choice; and 2) `SELECT` statement gives rise to a choice whose outcome depends upon the presence of tokens at port places. Since the processes are sequential and the places modeling the program counter are different, the choice of which transition to fire is up to the process code in the first case, and up to the scheduler (based on the availability of tokens) in the second case.

2) Linking: After compilation, a PN is obtained for each process. Each PN has some dangling places representing the ports.

Linking combines all the PNs generated by compilation into a single one, by merging each pair of places corresponding to ports connected by a channel.

For each input (respectively, output) port connected to the environment, a source (respectively, sink) transition is connected to (respectively, from) the place corresponding to the port, where the weight of the arc denotes the size of data moved to (respectively, from) the port.

Fig. 2(a) depicts the PN obtained after compiling and linking the two processes specified in Fig. 1. The dotted line separates the PN fragments corresponding to each process. Linking is performed by merging the ports of channel `DATA` into the place with the same name.

III. 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 $(\text{PN} + \text{scheduler})$ proceeds from state to state by firing fireable transitions. Formally:

Definition 6 (Sequential Schedule): Given a PN $N = (P, T, F, M_0)$, 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 $\mu(s_0) = M_0$.[3]
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) For each state $s \in S$, there is a path $s \xrightarrow{\sigma} s' \xrightarrow{t}$ for each uncontrollable $t \in T_{\text{S}}^{\text{u}}$.

Property 2 implies trace containment between 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. Finally, the existence of the path in property 4, coupled with a weak fairness assumption, ensures 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 uncontrollable 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. Note that, by definition, all uncontrollable source transitions belong to a single FCS.

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

1) The environment makes a move by firing any of the uncontrollable source transitions.
2) 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 one of the uncontrollable 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*

[2]Without loss of generality, in order to achieve better clarity, figures in the paper may not show this clustering.

[3]This mapping is required in order to enable the same state to be visited multiple times with different termination criteria, as will be discussed in detail in Section IV-A.

postpone these by choosing some other enabled transitions to fire.

The goal of the game is to process any input from the environment (property 4) while keeping the traversed space, and hence the amount of memory required to implement the communication buffers, 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 to allow the scheduler to fire all nonsource transitions, the schedule is an upper approximation of the set of states visited during real-time operation. This is due to the fact that the scheduler is constructed taking into account the worst possible conditions, since it has no knowledge about the correlations discussed above among environment inputs and among data-dependent choices.

Note that a schedule may include transitions of processes, even if they are not reachable from any uncontrollable source transition through directed paths in the PN. This is because the scheduler may freely decide to fire such transitions as long as they are enabled at given markings. While such transitions may be unnecessary to constitute schedules, and thus may be redundant in the resulting behavior, there are cases where such transitions must be fired in any schedule. For example, if one process autonomously generates a sequence of random numbers that is written to a channel to another process, while the second process combines the sequence with a data stream from the environment, then the transitions of the first process are not structurally reachable from the uncontrollable source transition that triggers the second process. However, any sequential schedule of this system will include the transitions of the first process, because they are necessary for the second process to fire the transition for the operation that reads from the channel, which in turn must be included to constitute a schedule. Our scheduling procedure includes such transitions, using the t-invariant heuristics described in Section IV.

The notion of sequential schedule is illustrated in Figs. 4 and 5. Fig. 4 shows two nonschedulable specifications and parts of their reachability spaces.

The impossibility of finding a schedule for the PN in Fig. 4(a) stems from the inability of a scheduler to control the firing of the uncontrollable source transitions (from now on uncontrollable transitions are depicted in figures as shadowed boxes). A cyclic behavior in this PN is possible only with correlated input rates of transitions a and b, as shown by the corresponding part of a reachability graph. On the other hand, the PN in Fig. 4(b) is nonschedulable because of the lack of control on the outcome of free-choice resolution for place $p1$. Bounding the reachability space would require alternation in the firing of transitions a and b. Note that the relative firing rates of the two transitions could also be chosen by the scheduler, following the approach of [12], when input transitions are controlled by the scheduler.

Fig. 5(a) presents an example of arbitration with two processes competing for the same resource (modeled by a token in choice place p_0). The schedule for this specification is given in Fig. 5(b), where await states are shown by shadowed rec-

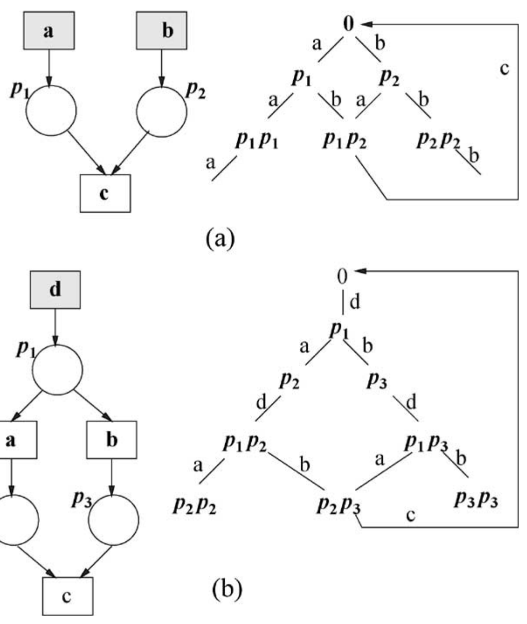


Fig. 4. Nonschedulable PNs: all source transitions are of the uncontrollable type.

tangles.[4] Note that the scheduler makes a “smart” choice about which one among the concurrently enabled transitions a, d, or f fires in state $\{p_4, p_5\}$, by first scheduling transition f to release the common resource (token in p_0) as quickly as possible. This helps to represent the schedule as a composition of two independent components (see Section III-B).

A. Reactive Schedules

In general, a system needs to be initialized before entering the normal (repetitive) mode of operation. Initialization is usually not part of system functionality, because it is performed only once, and during it, the system response to the environment is often irrelevant. The following definitions help in distinguishing between the initialization and reactive parts of a schedule.

Definition 7 (Initialization Part): The initialization part of a sequential schedule $Sch = (S, T, \rightarrow, s_0)$ is the restriction of Sch to the unique maximal connected set of states S^i reachable from s_0 and not containing any await state.

Intuitively, the initialization part is obtained as a maximal schedule prefix before reaching await states. If there is a unique await state s_0^{r} that terminates the initialization part $(enter(S \setminus S^i) = s_0^{\mathrm{r}})$ then we will call a schedule *well initialized*. The rest of this section considers only well-initialized schedules.

Definition 8 (Reactive Part): The reactive part of a well-initialized sequential schedule $Sch = (S, T, \rightarrow, s_0)$ is a transition system $Sch^{\mathrm{r}} = (S^{\mathrm{r}}, T, \rightarrow, s_0^{\mathrm{r}})$ obtained as the restriction of Sch to the set of states S^{r} reachable in Sch from $s_0^{\mathrm{r}} = enter(S \setminus S^i)$.

Fig. 6 shows examples of different shapes of the initialization and reactive parts for well-initialized schedules. Fig. 6(a)

[4]When no two states of a schedule are mapped to the same marking, with some abuse of notation, we will not distinguish between schedule states and PN markings.

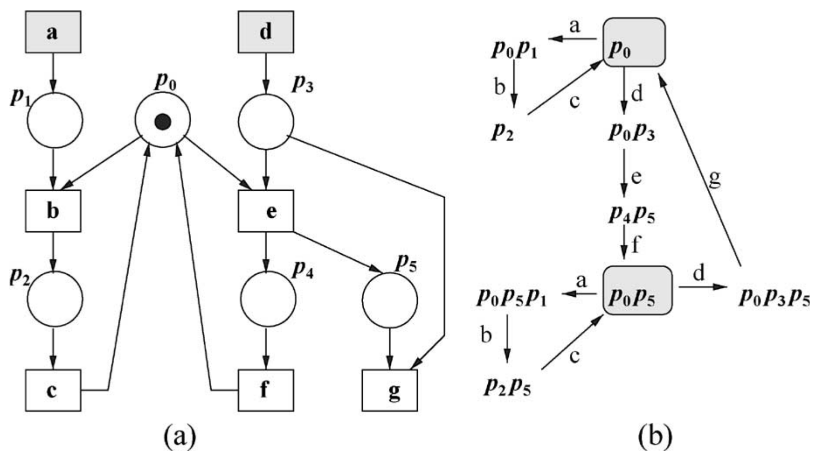


Fig. 5. Processes with arbitration: both source transitions are of the uncontrollable type.

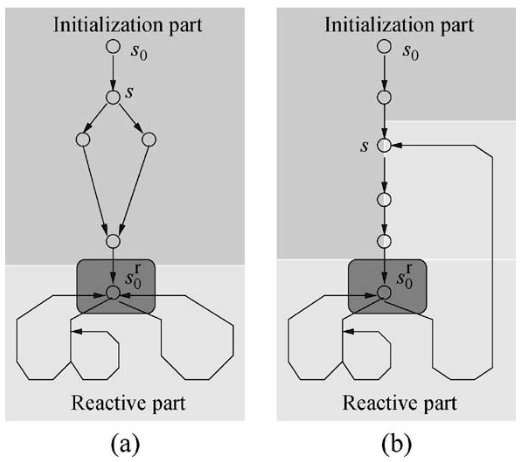


Fig. 6. Well-initialized schedules.

$$\begin{tikzcd}
\text{Original PN} \arrow[r, "\text{Scheduling}"] \arrow[d, "\text{Change of initial marking}"'] & \text{Well-initialized schedule} \arrow[d, "\text{Abstracting initialization}"] \\
\text{Reactive PN} \arrow[r, "\text{Scheduling}"] & \text{Reactive part}
\end{tikzcd}$$

Fig. 7. Reactive objects in scheduling.

illustrates that the well-initialization conditions do not require a fully deterministic initialization behavior (see, e.g., the choice state s), but they do demand nondeterminism to be resolved before reaching the reactive part (state s_0^{r}). Fig. 6(b) shows that the initialization and reactive parts do not partition the set of schedule states, because they can partially overlap (see the part of TS from state s up to the first await state s_0^{r}).

Definition 8 allows us to remove the unnecessary details of initialization from a schedule, and to concentrate solely on the reactive mode of system operation. For a PN $N = (P, T, F, M_0)$ with well-initialized schedule Sch, abstracting the initialization behavior translates to changing the initial marking M_0 to marking M_0^{r}, corresponding to the entry state s_0^{r} of the reactive part of Sch ($M_0^{\mathrm{r}} = \mu(s_0^{\mathrm{r}})$). Such PN $N = (P, T, F, M_0^{\mathrm{r}})$ is called a *reactive* PN. From a reactive PN, it is possible to derive the reactive part of a schedule by imposing additional constraints on Definition 6.

Definition 9 (Reactive Schedule): Given a PN $N = (P, T, F, M_0^{\mathrm{r}})$, a *reactive schedule* of N is a sequential schedule in which only source transitions are fireable in the initial state $s_0 : \mu(s_0) = M_0^{\mathrm{r}}$.

The correspondence between the original PN and its sequential schedule and their reactive counterparts is illustrated by Fig. 7.

B. Single-Source Schedules: Rationale

As discussed in Section I, if the uncontrollable inputs are totally independent, a schedule may be given as a set of *tasks*, where each task defines a schedule for a particular uncontrollable input. We call such a task single-source schedule (SSS).

An SSS is a reactive 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, an SSS gives a projection of the scheduler activity in which only one source transitions is fireable.

The advantages of SSSs over a single reactive schedule can be summarized as follows:

1) Lower complexity for the generation of SSSs. The size of a monolithic reactive schedule can be exponentially larger than the size of the set of SSSs.
2) SSSs give a natural decomposition of a sequential schedule that is beneficial for implementation as interrupt service routines on an RTOS.
3) A scheduler that behaves according to SSSs provides a uniform response for all 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.

However, we can enjoy this advantage only if the set of SSSs indeed implements the system execution. In other words, we need to ensure that the composition of the SSSs fulfills the properties of *reactive schedule* (see definition 9). In the following two subsections, we present a formal definition of an SSS and provide a condition under which the set of SSSs can be used as a schedule of the system, instead of a monolithic reactive schedule.

C. Single-Source Schedules: Definition and Composition

Definition 10 (Single-Source Schedule): Given a reactive PN $N = (P, T, F, M_0^{\mathrm{r}})$, a *single-source schedule* of N with respect to uncontrollable source transition $a \in T_{\mathrm{S}}^{\mathrm{u}}$ is a reactive schedule of a net in which all uncontrollable transitions except a are deleted $(N \setminus (T_{\mathrm{S}}^{\mathrm{u}} \setminus \{a\}))$.

The sequential composition of a set of SSSs is defined as follows. 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 fired. 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 11 (Sequential Composition): Let $N = (P, T, F, M_0^{\mathrm{r}})$ be a reactive PN and $\mathcal{X} = \{\mathsf{SSS}(t_i) = (S_i, T_{t_i}, \rightarrow_i, s_{0_i}) \mid t_i \in T_{\mathrm{S}}^{\mathrm{u}}\}$ 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:

1) $s_0 = (s_{0_1}, \ldots, s_{0_k})$;
2) $S \subseteq S_1 \times \cdots \times S_k$ 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;
3) for every state $s = (s_1, \ldots, s_k)$:
   a) 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 $\mathsf{SSS}(t_i)$;
   b) if s is not an await state, there is one, and only one,[5] 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 $\mathsf{SSS}(t_i)$.

Fig. 8 depicts the sequential composition of two SSSs obtained from the PN in Fig. 5. The shadowed circles correspond to await states. Initially, both SSSs are in await states. Thus, only uncontrollable 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 $\mathsf{SSS}(d)$ until a new await state is reached (state 3 of $\mathsf{SSS}(d)$). 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.

Definition 12 (Sequential Independence): Given a reactive PN $N = (P, T, F, M_0^{\mathrm{r}})$, a set of single-source schedules $\mathcal{X}$ is said to be *sequentially independent* if its sequential composition is isomorphic to a reactive schedule of N.

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

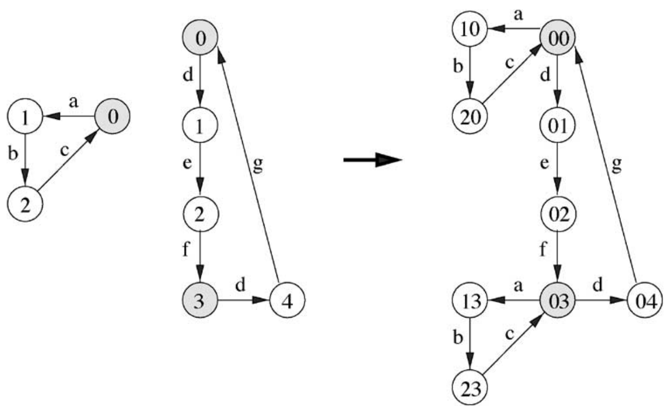


Fig. 8. Two single-source schedules and their sequential composition.

One can easily check that the sequential composition in Fig. 8 is isomorphic to the reactive schedule in Fig. 5(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 reactive schedule implies the existence of SSSs (once a reactive schedule has been obtained all SSSs can be immediately derived by using the subgraphs in which only one source transition fires). Moreover, Definition 12 indicates that sequential independence for a set of SSSs is a sufficient condition for the existence of a reactive 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.

D. Checking Sequential Independence

Given a reactive PN 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 reactive schedule of N, according to Definition 9.

This approach is computationally expensive because it requires the explicit derivation of the composition of SSSs.

We next propose an alternative way for checking the independence of SSSs that does not require the calculation of their composition. Let us consider the case in which the SSSs are not independent, resulting in a failure to find an isomorphic reactive 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 PN. For nonindependent A and Sch, 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 PN cannot simulate the sequential composition of SSSs. Fig. 9 shows the structure of the paths, where shadowed circles denote await states.

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 being disabled 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, $\mathsf{SSS}(t_i)$

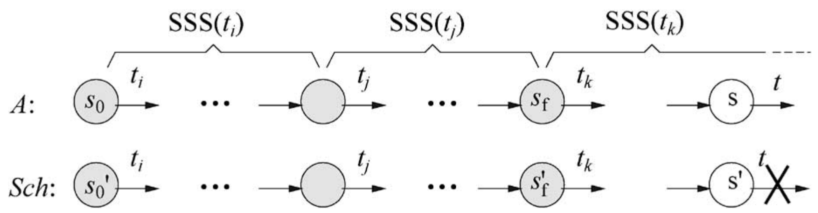


Fig. 9. Matching SSS composition with a reactive schedule.

and $\mathsf{SSS}(t_j)$ must consume tokens from some place p in the preset of t.

The following hierarchy of notions is used for the formulation of independence via marking properties.[6]

1) For $\mathcal{X} = \{\mathsf{SSS}(t_i) \mid t_i \in T_S^u\}$ and given place p
   a) for $\mathsf{SSS}(t_i)$ with set of states S_{t_i} and set of await states $S_{t_i}^a$:
      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 the initial state of $\mathsf{SSS}(t_i)$ and state s.
      ii) 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.
      iii) let $await\_change(p, t_i) = \max_{s \in S_{t_i}^a} change(p, s)$, i.e., the maximal change in token count for place p in markings corresponding to the await states of $\mathsf{SSS}(t_i)$ with respect to the initial marking.
   b) Let $worst\_change(p, t_i) = \sum_{t_j \neq t_i, t_j \in T_S^u} await\_change(p, t_j)$, i.e., the sum of $await\_change$ for all SSS except for $\mathsf{SSS}(t_i)$.

Here is the semantics of the introduced notions.

1) $\mathsf{SSS}\_change(p, t_i)$ 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 postset of p.
2) $await\_change(p, t_i)$ 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 the scheduler switches among tasks (SSSs), 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.
3) $worst\_change(p, t_i)$ 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 $worst\_change(p, t_i)$.

Fig. 10 depicts a diagram that represents the changes on a place p produced by three single-source schedules. The horizontal axis represents the evolution of the schedule through different states, and the black dots represent await states. The solid line represents a partial trace of the system, in which all schedules but one ($\mathsf{SSS}(t_1)$ and $\mathsf{SSS}(t_2)$) are in await states, whereas the other schedule ($\mathsf{SSS}(t_3)$) is in an arbitrary state. For the trace to be fireable, we need the condition that the accumulated changes on p do not lead to a negative number of tokens.

The following theorem establishes the bridge between the sequential independence of SSSs and the firing rule of PNs, when the schedules are executed.

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

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

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 reactive schedule isomorphic to the composition of $\mathcal{X}$. Abusing notation, we will make no distinction between the states of the composition and the corresponding reactive schedule.

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, it follows that by reaching s in the composition, the corresponding marking for place p equals $M_0^r(p) - worst\_change(p, t_i)$. 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 $M_0^r(p) - worst\_change(p, t_i) - \mathsf{SSS}\_change(p, t_i) < 0$, then in the reactive 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 reactive schedule.

$\Leftarrow$. Suppose that inequality (IE.1) is satisfied but $\mathcal{X}$ is not sequentially independent. In the set of all reactive schedules, let us choose the schedule Sch that is isomorphic to the largest subpart of the sequential composition A. That is, if a mismatch like the one in Fig. 9 is found by simulating Sch and A, then there does not exist any other reactive schedule

[6]For convenience, we assume PNs without self-loops. This does not impose any constraint on the class of specification that can be handled, because any PN with self-loops can be equivalently transformed into a self-loop-free PN by inserting silent transitions.

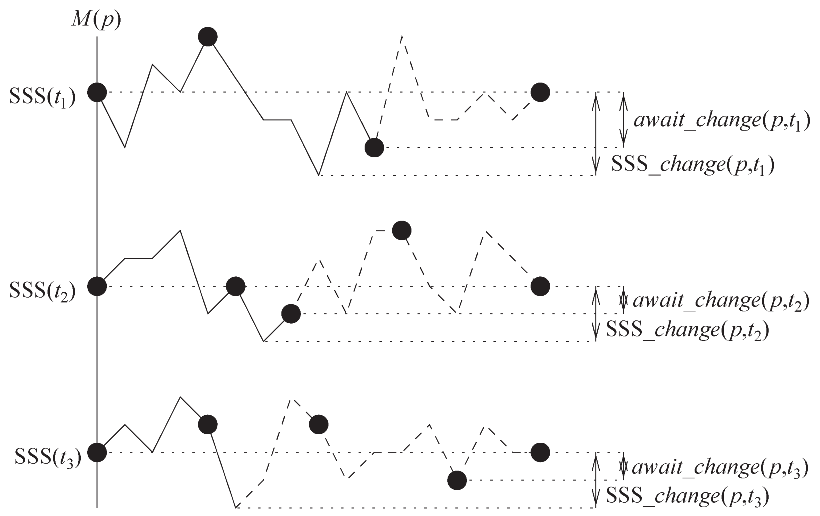


Fig. 10. Changes on the marking of p produced by the composition of sequential schedules.

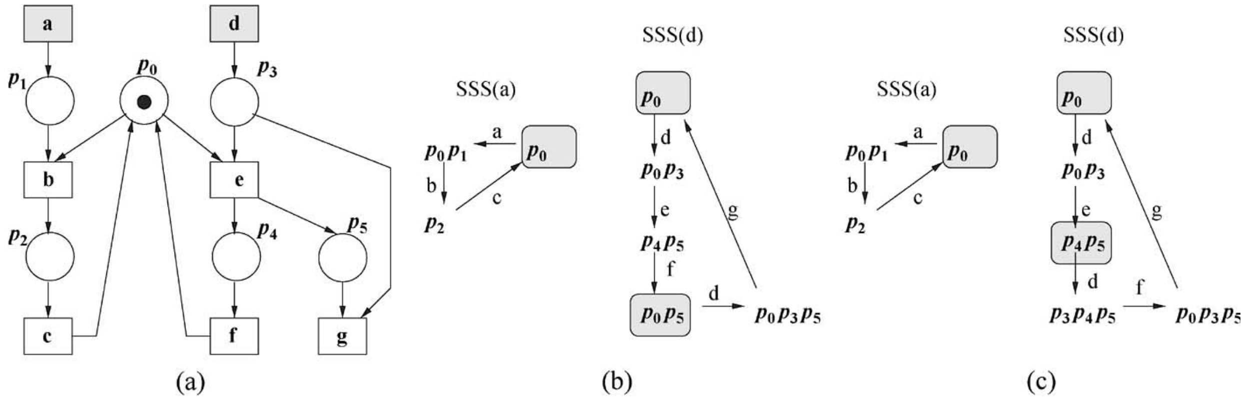


Fig. 11. Process with arbitration and its single-source schedules.

with a state s'' isomorphic to s and capable of firing transition t. Let us rearrange the sequence in Fig. 9 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 place p in the marking corresponding to s_{f} is larger than $M_0^{\mathrm{r}}(p) - worst\_change(p, t_i)$. By definition, the execution of $\mathsf{SSS}(t_i)$ cannot reduce it by more than $\mathsf{SSS}\_change(p, t_i)$. 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. 9 is impossible. ■

One could thus derive from Theorem 1 a simple sufficient condition for checking sequential independence.

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 $\forall p : M(p) \geq M_0^{\mathrm{r}}(p)$.

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

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

We will illustrate the suggested approach for checking the independence of SSSs by using the example of processes with arbitration in Fig. 5. Two different sets of single-source schedules for this example are shown in Fig. 11(b) and (c).

The only place shared by both $\mathsf{SSS}(a)$ and $\mathsf{SSS}(d)$ is 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. 11(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 nonawait 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 $M_0^{\mathrm{r}}(p_0) = 1$, 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. 8).

Reversing the order of firing for transitions d and f in $\mathsf{SSS}(d)$ from Fig. 11(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 reveals the dependence between $\mathsf{SSS}(a)$ and $\mathsf{SSS}(d)$ from

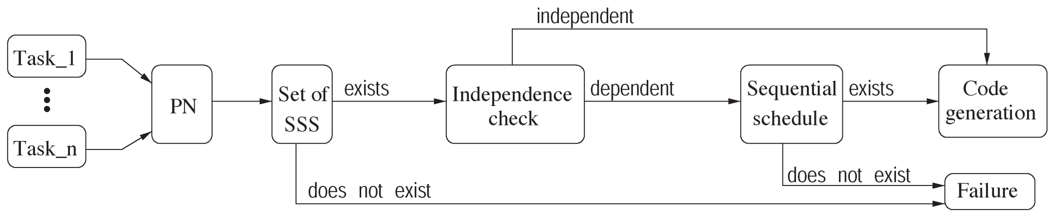


Fig. 12. Design flow for quasi-static scheduling.

Fig. 11(c). Note that the same result could be immediately concluded by considering the await states of $\mathsf{SSS}(d)$ and applying Corollary 1.

From the above example, it follows that from the very same specification one can obtain both independent and dependent sets of SSSs. In case an independent set exists (this does not happen for all PNs), it would be desirable to find it. However, this is difficult because in the worst case it requires the exhaustive exploration of the concurrence in all SSSs. Therefore, for practicality, we suggest using a "try and check" approach in which a set of SSSs is derived, and if they are not independent, a reactive schedule is immediately constructed (if possible). This design flow for scheduling is illustrated by Fig. 12.

E. Termination Criteria

Sequential schedules are derived by exploring the reachability graph of a PN with source transitions. Unfortunately, this graph is infinite.

A possible approach to tackle the problem of dealing with an infinite reachability graph is to explore only a finite subset defined heuristically or by some formal criteria. One enumerative formal criterion that provides a *semidecision* procedure (that succeeds if the PN is schedulable) can be derived following the approach in [17], by initially setting some bounds on all places based on the structural properties of the PN, and increasing the bounds every time a deadlock due to capacity is reached. However, this approach needs to exhaustively analyze the reachability space for each set of bounds if a schedule does not exist within them, and it is not applicable in practice because of the prohibitively large reachability spaces even for the initial bounds.

Next, we discuss *conservative* heuristic approaches to prune the exploration of the reachability space while constructing a schedule, which allow our approach to always terminate. 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 moves towards directions that are qualified as *nonpromising*, i.e., where 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 PN and 2) markings are qualified as irrelevant during the exploration of the state space if they both cover some preceding marking and exceed the calculated bounds. Note that the property of irrelevance, as defined below, is not local and depends on the prehistory of the marking, i.e., on the sequence of markings visited before it from the initial one.

Definition 13 (Place Degree): The degree of a place p is defined as

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

Place degree intuitively models the "saturation" of p. If the token count of p is $\max_{t\in {}^\bullet p} F(p,t)$ or more, then adding tokens to p cannot help in enabling output transitions of p. The firing of a single input transition of p can add at most $\max_{t\in {}^\bullet p} F(t,p)$ tokens, which gives the expression for place degree shown in Definition 13.

Definition 14 (Irrelevant Marking): A marking M is called *irrelevant* with respect to the reachability tree rooted in initial marking M_0, if the tree contains marking M_1 such that:

1) M is reachable from M_1;
2) no place has more tokens in M_1 than in M; and
3) 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. 13 illustrates the crucial difference between the approaches based on predefined place bounds and irrelevant markings.

The maximal place degree in the PN of Fig. 13(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 bounds for places should be chosen to at least exceed place degrees. Suppose that, based on this rationale, the bounds are chosen as the 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 predefined place bounds, any marking that has more than three 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 four tokens in place $p2$ (see the part of the reachability graph shown in Fig. 13(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 towards 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, the scheduler must choose which transitions,

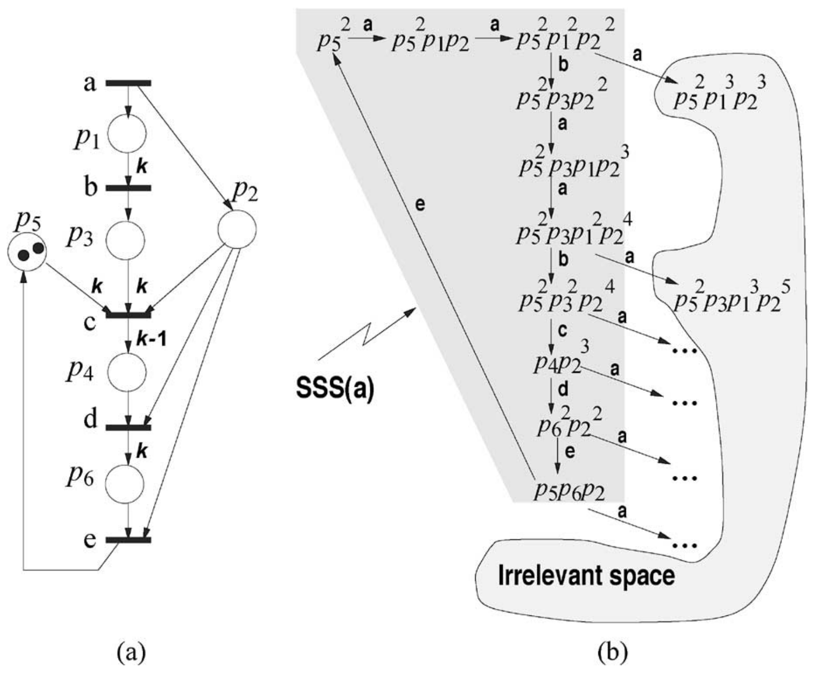


Fig. 13. Constraining the search space by irrelevance criterion.

a or b, to fire from the enabled set. The firing of a however produces 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 twice, 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. 13(b)]. Finally, this procedure succeeds and finds a valid SS schedule.

Though pruning, the search using irrelevance seems a more justified criterion than using place bounds (as used, e.g., in [13], [19]), 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 the 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 our conjecture that the irrelevance criterion is exact for PNs with choice places that are either unique or free choice. Note that FlowC specifications without the SELECT operator belong to this class. However, we have so far been unable either to prove the exactness of this criterion, or to find a counterexample. This issue is left open for the moment.

IV. Algorithm for Schedule Generation

In this section, we present an algorithm for computing a sequential schedule of a given PN. This algorithm can also be used to compute a single-source schedule for a source transition t_i, if it takes as input a reactive PN in which all the source transitions except t_i are deleted (see Definition 10). Finally a sequential program is generated from the resulting schedule by the procedure described in Section IV-C.

A. 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 function $\mathtt{EP}(r,r)$, shown in Fig. 14(a). If this function returns successfully, a post-processing step 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 $[v,w]$.

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 nonleaf 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. FCSs are associated so that the conditions given in the definition of sequential schedules (Definition 6) are satisfied, which will be elaborated next.

```
function EP(v, target)
   // returns (status, ap, ep)
1  $ap \leftarrow 0$, $ep \leftarrow$ UNDEF, $FCS(v) \leftarrow \phi$;
2  if(termination conditions hold)
3      return (0, 0,UNDEF);
4  if($\exists u : u < v$ and $\mu(u) = \mu(v)$)
5      return $(1, 0, u)$;
6  for(each FCS $F$ enabled at $\mu(v)$)
7      if($F = T_S$) $current\_target \leftarrow v$;
8      else $current\_target \leftarrow target$;
9      ($status$, $ap_F$, $ep_F$) $\leftarrow$
              EP_FCS($F$, $v$, $current\_target$);
10     if($status = 0$) continue;
11     if($ap_F = 1$)
12         $FCS(v) \leftarrow F$, return $(1, 1, ep_F)$;
13     if($ep_F \leq current\_target$)
14         $FCS(v) \leftarrow F$, return $(1, 0, ep_F)$;
15     if($ep_F < ep$)
16         $FCS(v) \leftarrow F$, $ap \leftarrow ap_F$, $ep \leftarrow ep_F$;
17 if($FCS(v) = \phi$) return $(0, ap, ep)$;
18 else return $(1, ap, ep)$;
```

(a)

```
function EP_FCS($F$, $v$, $target$)
       // returns ($status$, $ap_F$, $ep_F$)
1  $ap_F \leftarrow 0$, $ep_F \leftarrow$ UNDEF,
2  $current\_target \leftarrow target$;
3  for(each transition $t$ of $F$)
4      create a node $w$ and an edge $[v, w]$;
5      $T([v, w]) \leftarrow t$;
6      $\mu(w) \leftarrow$ the marking obtained by
              firing $t$ at $\mu(v)$;
7      ($status, ap, ep$) $\leftarrow$ EP($w$, $current\_target$);
8      if(status = 0) return $(0, ap_F,$UNDEF$)$;
9      if($ap = 1$ or $FCS(w) = T_S$)
10         $ap_F \leftarrow 1$, $current\_target \leftarrow v$;
11     if($ap_F = 0$ and $v < ep$)
12         return (0, 0, UNDEF);
13     if($ep \leq v$) $ep_F \leftarrow \min(ep_F, ep)$;
14     if($ep_F \leq target$) $current\_target \leftarrow v$;
15 return $(1, ap_F, ep_F)$;
```

(b)

Fig. 14. The two main functions called in computing a sequential schedule.

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. 14(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 called an *entry point* of v, which is recursively defined for nodes and FCSs enabled at their markings. If v is a leaf and has a proper ancestor with the same marking, the ancestor is the entry point of v. Otherwise, $ep(v)$ is an entry point of some FCS enabled at $\mu(v)$. An entry point ep_F of an FCS F is defined if the following two conditions hold. First, for each transition t of the F, 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 $ep(w) \leq v$. In this case, ep_F is the minimum among the $ep(w)$ 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, ep_F is undefined. If there is no FCS that satisfies these conditions, or ep_F is undefined for each FCS that satisfies the conditions, $ep(v)$ is undefined and 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 $\mu(v)$ with the resulting marking equal to that of the node given by $ep(w)$. 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 function EP_FCS for each FCS enabled at $\mu(v)$. If EP succeeds at the root r, we call the postprocessing step to create a schedule and terminate. Otherwise, we report no schedule and terminate.

The algorithm can use any termination condition at line 2 of Fig. 14(a), if it can be evaluated on the tree being constructed and the nodes v and $target$. Such a condition includes the one given in Section III-E, as well as the one that specifies bounds on the number of tokens at the places.

Fig. 15 illustrates the algorithm for the PN given in Fig. 2. Fig. 15(a) shows the tree obtained just before the postprocessing, when the algorithm is applied to the PN of Fig. 2. Suppose that we use bounds on marking of places as the termination condition, where we set a bound for each place equal to 1. The marking associated with each node is shown in thee parentheses adjacent to the name of the node. Fig. 15(b) presents the final schedule.

At each node we assume that the FCS shown is processed first, among those enabled at the marking. Suppose that the procedure has arrived at v_2. $\mathrm{EP}(v_2, r)$ is called at this node, which then calls $\mathrm{EP\_FCS}(\{t_3, t_2\}, v_2, r)$. EP_FCS creates a node v_3 for the transition t_3 and calls $\mathrm{EP}(v_3, r)$. The only FCS enabled at the marking of v_3 is the set T_{S} of the source transitions. Thus EP sets $current\_target$ to v_3 and calls $\mathrm{EP\_FCS}(T_{\mathrm{S}}, v_3, v_3)$. EP_FCS then processes each of the two source transitions.

Consider transition T_{in}, and suppose that the procedure has arrived at the point where v_7 is created in EP_FCS. The $target$ is still v_3, and thus $\mathrm{EP}(v_7, v_3)$ is applied. EP calls EP_FCS with the FCS $\{t_8, t_9\}$. EP_FCS then creates a node v_8 for the transition t_9 and calls $\mathrm{EP}(v_8, v_3)$. Since the marking of v_8 is equal to that of v_2, EP returns (1, 0, v_2). It then processes the other transition t_8, for which EP returns (1, 0, v_2).

Suppose now that the procedure has come back to the node v_3, at which $\mathrm{EP\_FCS}(T_{\mathrm{S}}, v_3, v_3)$ returns (1, 0, v_2). Since

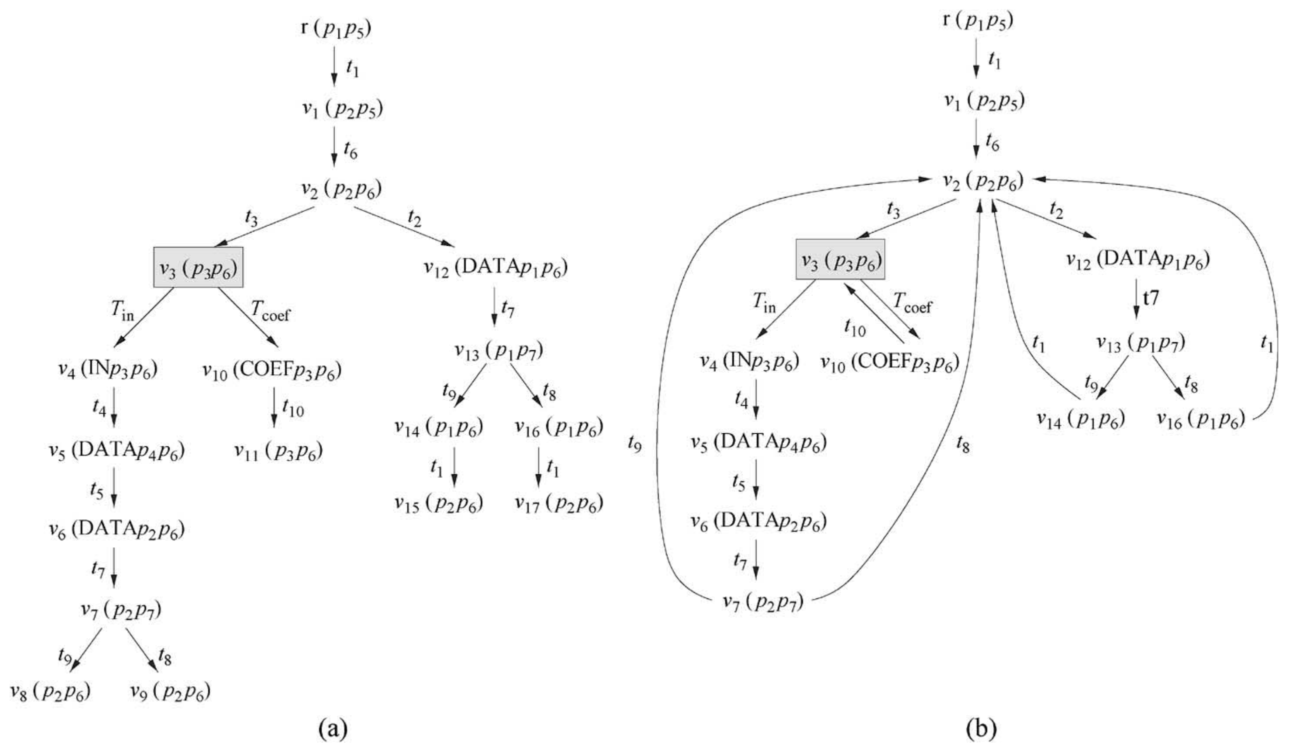


Fig. 15. Scheduling tree and final schedule for the PN of Fig. 2.

$current\_target$ had been set to v_3 in EP(v_3, r), $v_2 \leq current\_target$ holds. Therefore, EP(v_3, r) immediately returns (1, 0, v_2) to EP_FCS$(\{t_2, t_3\}, v_2, r)$. $FCS(v_3)$, the FCS assigned at v_3, is T_S, and therefore EP_FCS$(\{t_3, t_2\}, v_2, r)$ sets ap_F to 1 and $current\_target$ to v_2. It then continues for the transition t_2 by calling EP(v_{12}, v_2). It will return (1, 0, v_2) and EP_FCS$(\{t_2, t_3\}, v_2, r)$ returns (1, 1, v_2). These values are propagated to the root, and are finally returned by EP(r, r). The postprocessing step is then called, which deletes the nodes v_8, v_9, v_{11}, v_{15}, and v_{17}, and creates cycles as shown in Fig. 15(b).

The postprocessing step consists of two parts. First, we retain only those parts of the 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 can be shown that this algorithm always finds a schedule, if there exists one in the space defined by the termination conditions employed in EP.

Note that the resulting graph may have more than one node with the same marking. Further, FCSs associated with these nodes may not be the same. The freedom of associating different FCSs at nodes with the same marking allows the algorithm to explore a larger solution space and thus classify more PNs as schedulable. However, it is also possible to force the graph to have at most one node for a given marking, and thus require less memory at scheduling time, by slightly changing the algorithm. Specifically, in EP_FCS, before creating a node w for a given transition t, we first compute the marking obtained by firing t at $\mu(v)$, and check if there is a node y in the graph such that $\mu(y)$ is equal to the marking and $status(y) = 1$. If so, instead of creating a node w, we create an edge $[v, y]$, set $(status, ap, ep)$ to $(1, ap(y), ep(y))$, and continue to the next transition.

B. *Sorting Enabled* FCS*s*

The pseudocode for EP shown in Fig. 14(a) processes all the enabled FCSs before completing the function, unless there is one with which either $ap_F = 1$ or $ep_F \leq current\_target$ holds, i.e., with this FCS, v will either have a path to an await node, or $ep(v)$ will be an ancestor of $current\_target$. In practice, there is an FCS for which neither of the conditions holds, but its entry point ep_F is a proper ancestor of v. Although the pseudocode does not immediately return with such an FCS, the FCS can lead to a valid schedule if there is a node between ep_F and v at which there is an enabled FCS, whose entry point is an ancestor of $current\_target$. The algorithm can be implemented to take advantage of this possibility, with potential cost of backtracking. Specifically, if EP_FCS has returned (1, ap_F, ep_F) for an enabled FCS F with $ep_F < v$, EP immediately returns (1, ap_F, ep_F). This will let the procedure return to EP processing the node ep_F, i.e., the end of line 10 of the pseudocode where EP_FCS returns for the FCS being tried in the EP. If the returned values from EP_FCS do not satisfy the conditions to let EP further return, then it recursively goes back to its children to process FCSs for which EP_FCS has not been called. This implementation often finds a schedule more quickly than the original algorithm. We call this heuristic *speculative*-FCS, and compare its effectiveness to that of the original algorithm in Section V.

In general, the number of nodes created in the algorithm depends on the order of FCSs explored in EP. Although the ordering does not influence the worst case search space or run time of the algorithm, some orderings lead to a schedule sooner than others. We employ a heuristic approach of sorting the FCSs for this reason, which also provides us with a sufficient nonschedulability condition, i.e., if the condition holds, we can immediately terminate the procedure, reporting no schedule. It tries to find a short sequence of transitions such that if the sequence is fired from the node being processed in EP, a marking associated with some ancestor of the node can be obtained. Such an ancestor becomes a candidate to be an entry point for the node. We represent the sequence using a vector of nonnegative integers, which we call a *promising vector*. Its positions correspond to transitions, and each entry represents the number of occurrences of the corresponding transition in the sequence. It is taken as additional input by both EP and EP_FCS, where it is initially set to the null vector. When it is null, EP computes one, sorts the FCSs, and then passes it to EP_FCS. In EP_FCS, for each transition to be processed, the integer at the corresponding position in the vector is decreased by 1, and the result is passed to EP. If the resulting integer is negative, the null vector is passed.

We use t-invariants [16] to find such a vector. A t-invariant is a vector of nonnegative integers that solves the system of homogeneous marking equations $Cx = 0$, where C is the incidence matrix of the PN. The incidence matrix is a matrix of $|P| \times |T|$ integers defined as $C_{ij} = F(t_j, p_i) - F(p_i, t_j)$, where p_i and t_j correspond to the ith place and jth transition, respectively. A t-invariant represents sequences in which the number of occurrences of the jth transition is equal to the integer at the jth position of the t-invariant. Each such sequence has the property that if it can be fired from a marking M, the marking obtained after the firing is also M. We call a nonnegative basis of the homogeneous marking equations a basis of t-invariants, which can be computed by existing methods [16]. Since a schedule does not exist if there is no basis of t-invariants, if the algorithm identifies this case, it terminates immediately without applying the function EP.

Suppose that a basis of t-invariants is found. The heuristic first finds a subset of the basis with the property described in Theorem 2 below, then sorts FCSs using it. The property is related to the t-invariant being enabled. A known problem with t-invariants is that it is in general difficult to identify whether a t-invariant is enabled at a given marking, where we say that a vector of transitions is enabled if some sequence of transitions represented by the vector is enabled. For our case, however, due to the structure of the PN generated from a FlowC specification, a necessary condition can be obtained for an invariant to be enabled. For the sake of simplicity, we assume that the specification does not contain SELECT statements, although the extension to handle them is straightforward. To describe the condition, let us introduce some terminology. An FCS is said to be *pseudoenabled* at a marking M, if some transition of the FCS has a predecessor place p such that p is marked at M and p does not correspond to an FIFO queue in the original network specified in FlowC. This definition implies that the FCS does not contain a source transition and that it originates from the code of a single FlowC process. Further, we say that a process *appears* in a vector, if there exists a transition in the process that appears in the vector. Then the following theorem shows the necessary condition.

Theorem 2: For any vector of transitions enabled at a reachable marking M in the PN obtained from a FlowC specification without SELECT, and for any pseudoenabled FCS at M, if the process of the FCS appears in the vector, the FCS has a transition that appears in the vector.

Proof: Suppose, for the sake of contradiction, that there exists a pseudoenabled FCS at M, say F, such that its process appears in the vector, but no transition of F appears in the vector. Let t_1 be a transition in the process that appears in the vector. By definition, t_1 does not belong to F. Since the specification does not have SELECT statements, the PN fragment created for the process from the FlowC specification has exactly one pseudoenabled FCS at M. This is because tokens model either data presence or program counters (one counter for each process) and for any place that does not model FIFO queues, its successors constitute a single FCS. This implies the following two statements. First, the FCS containing t_1 is not pseudoenabled at M. Second, if a sequence of transitions is enabled at M and the marking reached after the firing enables t_1, the sequence must contain a transition of F. For each sequence of transitions represented by the vector, since it does not contain any transition of F but contains t_1, the sequence is not enabled at M. It follows that the vector is not enabled at M, which is a contradiction. ■

Our heuristic finds a subset with minimal cardinality of the basis of t-invariants, such that the sum of the t-invariants in the subset satisfies this necessary condition at $\mu(v)$, where v is the current node being processed in EP. This problem can be formulated as a binate covering problem. Consider a matrix A, such that columns correspond to the invariants of the basis and rows correspond to pseudoenabled FCSs at $M(v)$. The row corresponding to FCS F has 0 at a given column, if the process of F appears in the invariant b corresponding to the column, but none of the transitions of F appears in b, 1 if F contains a transition that appears in b, and 2 otherwise. A subset of the columns of A is said to be a feasible solution of the binate covering problem, if for each row i of A, either there is no column j in the subset such that $A_{ij} = 0$, or there is a column j such that $A_{ij} = 1$. Informally, it means that for each pseudoenabled FCS F, if the process of F appears in some basis invariant included in a feasible solution, then there is a basis invariant in the solution that contains a transition of F. It follows that the subset of the basis given by a feasible solution of binate covering satisfies the necessary condition Theorem 2. We employ a method given in [15] that always finds a feasible solution, if it exists, while the cardinality of the subset is heuristically made minimum. In case no feasible solution exists, we set the subset to empty.

Once the subset is obtained, EP takes the sum of the invariants in the subset as a promising vector, and sorts FCSs enabled at $\mu(v)$ as follows. It favors most an FCS such that some of its transitions appear in the vector, and none of the transitions appears in the path from the root to v. This is followed by FCSs with some of their transitions in the vector. The rest is favored least. Ties can be broken by further using heuristics; for example, those that do not violate the termination conditions at their children are favored over those that do, or those with single transitions are favored over those with multiple transitions.

C. Code Generation

The code generation algorithm takes a graph of a sequential schedule and synthesizes code. In the sequel, we make no distinction between the graph and the schedule it represents. A direct translation of the schedule into code is possible, but it usually increases the code size, since different paths of the schedule may be associated with the same sequence of transitions, which yields a similar code segment. We thus perform optimizations to reduce the code size.

The code generation procedure that we have implemented uses two successive traversals of the schedule: the first identifies the initialization part and stops as soon as an await node is encountered; the second starts from the await node and identifies the reactive part. Both traversals follow the same approach divided in two steps: at first, a set of *code segments* is extracted from the schedule during the traversal; then, code for each segment is synthesized so that the behavior is correctly implemented.

1) Schedule Traversal: In the first step, the schedule is traversed in a depth-first manner to extract sequences of actions that are candidates to be shared in the generated code. In particular, the graph is divided into a set of *code segments*. A code segment is a directed rooted tree that associates an action with each edge, and a state to each node. A code segment is a schedule, in which await nodes can only be at the root, or at the leaves. It is not necessary for a code segment to have an await node. During code generation, code segments isomorphic to subtrees of the schedule are created.

As we shall see later, code segments represent uninterruptible sequences of actions. Since await nodes require the execution to be suspended, they are forbidden within a code segment. The goal of code generation is to find the minimum set of disjunct code segments such that:

1) an action in the schedule belongs to one, and only one, code segment;
2) each code segment is isomorphic to a set of subtrees of the schedule, such that each arc of each subtree has the same action as the corresponding arc of the segment;
3) the set covers the entire schedule, i.e., each node of the graph of the schedule is in a subtree, for which an isomorphic code segment exists.

The *state* of a node of a code segment is used to keep track of the flow of control, in case the same code segment is used to execute different paths in the schedule.

The first property above guarantees that we minimize the memory requirements, since code is maximally shared. The second property tells us that within a code segment there can only be local jumps (such as `if − then − else`), while global jumps from one code segment to another occur only at the leaves. This means that once the execution of a segment starts, it continues until a leaf is reached. Moreover, looking at the minimum set of code segments means that they are maximal, since if a segment is not maximal, then it can be made isomorphic to a larger subtree of the schedule, by merging it with another code segment that corresponds to the newly covered subgraph, without violating the above properties. Therefore, we minimize the performance loss due to jumping from one code segment to another. The third property guarantees that the entire behavior can be represented in terms of code segments.

The algorithm for the traversal is polynomial with respect to the size of the input graph. The traversal itself, being depth first, is linear, but to guarantee properties, we need to search already created code segments, at most once for each node of the graph. As the total size of the code segments is never greater than that of the initial graph, even a simple linear searching technique would make the overall algorithm quadratic in the size of the graph. Using a slightly more sophisticated search, the algorithm can be made $O(n \log n)$, where n is the number of nodes of the graph.

Fig. 16 shows an example of code segment extraction using the same schedule computed in Fig. 15(b), for the PN given in Fig. 2. Fig. 16(a) and (b) compute the initialization part, while Fig. 16(c) and (d) compute the reactive part.

During the traversal, nodes are flagged as visited, so that when a loop is found the traversal stops at that particular branch. However, a new node for the destination of the loop is created in the code segment that is being built. These flags are reset when the second traversal to identify the reactive part is started. As illustrated, code segments that are created at the beginning of the traversal may need to be split in order to guarantee that the above properties hold.

The algorithm recursively traverses the schedule to identify the code segments, stopping at each *await node* or when a transition that is already in a code segment is found: the first step for the initialization part creates code segment cs_1 shown in (a), where the pairs of a marking and an FCS are indicated in parentheses for the root and the leaf nodes. It stops at node u_3 as the corresponding node v_3 in the schedule is an await node, and at nodes u_{14} and u_{16} because the outgoing transition from v_{14} and v_{16} is t_1, already present in cs_1. The second step starts from v_{14}: it immediately recognizes that cs_1 should be split, because from t_1 you can either go to t_6, or to the choice between t_3 and t_2; therefore we get three code segments, as shown in Fig. 16(b). The third step starts from v_{16} and does not need to create any new code segment. No further traversal is needed because the frontier is an await node. The traversal for the reactive part starts from v_3 and creates a new code segment cs_4 rooted at u_3: it stops at u'_3 and u''_3 (an await node is reached), at v_{12} (the next transition is t_7, which is already present in cs_4), and u_8 (the corresponding node v_2 has already been visited and it is flagged); cs_4 thus needs to be split: the new cs_4, cs_5, and cs_6 that are generated are shown in Fig. 16(d). The traversal continues from v_{12} and does not need to create any new code segment until node v_{14} is reached, where cs_7 is finally created.

2) Code Synthesis: The second step generates the code to be used to implement a task. It does so by generating code for each code segment, adding a structure to jump from one code segment to another to reflect the original schedule. For this purpose, state variables are introduced for places and the markings in the tree are represented with them.

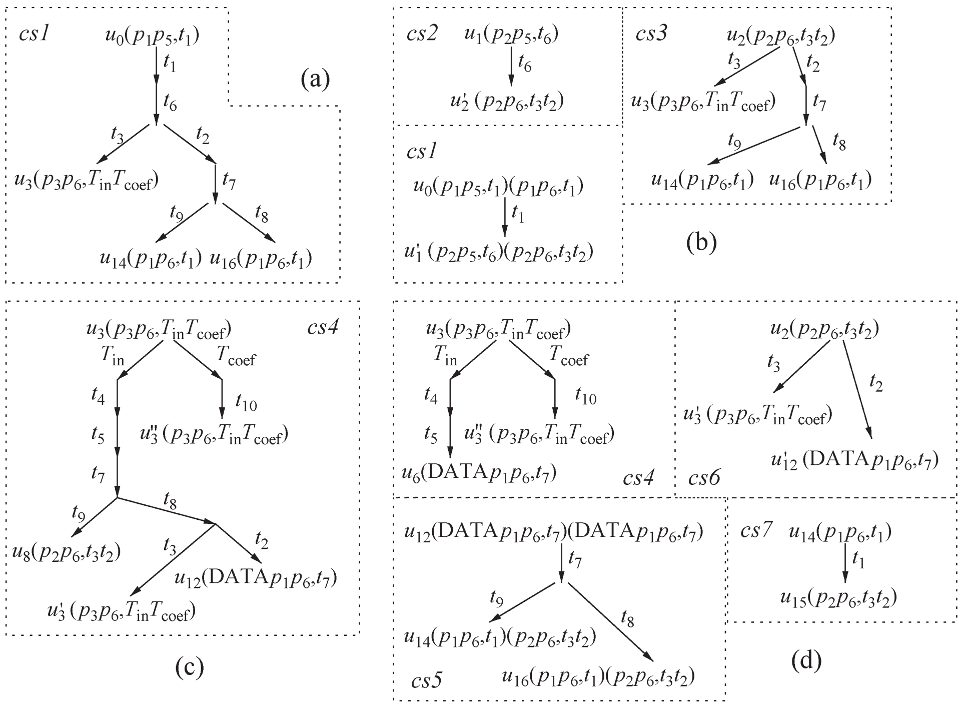


Fig. 16. Illustration of the code-generation algorithm for the schedule of Fig. 15(b).

The structure of a code segment is always the same and can be separated into three sections: *execution*, *update*, and *jump*.

1) *Execution*: It contains the real code for actions and data-dependent choices, taken from the original specification. It always starts with a unique label. The label is used to jump to the code segment.

   The graph for the code segment is traversed in a depth-first search manner. For each node, the code for the corresponding action is copied into the output file. If the node is a choice, then an `if − then − else` construct is generated using the condition specified in the node.

   When a leaf is reached, the *update* and *jump* sections are generated before going back in the traversal.[7]

2) *Update*: At each leaf of a code segment, the marking must be properly updated so that:
   a) the next code segment to execute in order to complete the computation can be correctly selected;
   b) the marking at the end of a sequence of code segments corresponds to the node in the schedule reached by the execution.

   If multiple code segments are traversed during a single reaction, each one of them is responsible for updating the state variable to reflect the change between the root node of the code segment, and the leaf that is reached. The sequence of these updates constitutes the global state change for that particular reaction.[8]

3) *Jump*: This section must find which code segment to call next, or should return if the reaction is finished. A switch construct on state variables is used to select a `goto` statement, which will cause the execution to jump to the unique label that identifies the next code segment. If the destination is an await node, then a `return` is generated instead of a `goto`.

Synthesis will therefore generate a set of functions: one for the initialization, and one for each input that should be served. They have no local variables and start with the first code segment (for the initialization part, it always corresponds to the root of the schedule; for the reactive part, it is always an await node), followed by all the others as needed by the jump sections. When the last code segment of a function is generated, the function is closed. These functions have just one entry point, but may have several exit points corresponding to all the leaves that perform a `return`.

Also, code for channels between processes that have been merged into a single schedule is generated. 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.

Fig. 17(a)–(c) show the code synthesized for the initialization and the reactive parts of the code segments illustrated in Fig. 16. Only places p_1 and p_5 are used to determine jumps, so they are the only ones whose marking is tracked in the code. Global variables are declared outside these functions (the code is not shown). From the schedule, it can be statically determined that place DATA holds at most one token, therefore a simple variable with assignments is used instead of a circular buffer.

[7]If there are choices in the code segment, then multiple update and jump sections are synthesized; however, if it can be statically determined that those sections are all identical, then they are merged after the closing of the `if − then − else`.

[8]As an optimization, although all state changes are tracked in the code segments, code is generated only for those that actually affect conditions to select jumps.

```
void Init() {
        p1 = 1; p5 = 1;
cs1:    sum = 0; i = 0;
        p1--;
        switch (p5) {
            case 1: goto cs2; break;
            case 0: goto cs3; break;
        }
cs2:    c = 1; j = 0;
        p5--;
        goto cs3;
cs3:    if (i < N) {
            return;
        } else {
            DATA = sum / N; d = DATA;
            if (j == N) {
                j = 0; d = d * c;
                WRITE(OUT, d, 1);
            } else {
                j++;
            }
            p1++;
            goto cs1;
        }
}
```

(a)

```
void Tin() {
cs4:    READ(IN, sample, 1); sum + = sample;
        DATA = sample; i++;
        goto cs5;
cs5:    d = DATA;
        if (j == N) {
            j = 0; d = d * c;
            WRITE(OUT, d, 1);
        } else {
            j++;
        }
        switch (p1) {
            case 1: goto cs7; break;
            case 0: goto cs6; break;
        }
cs6:    if (i < N) {
            return;
        } else {
            DATA = sum / N;
            p1++;
            goto cs5;
        }
cs7:    sum = 0; i = 0;
        p1--;
        goto cs6;
}
```

(b)

```
void Tcoef() {
cs4:    READ(COEF, c, 1);
}
```

(c)

```
void Init_optimized() {
        p1 = 1;
cs1:    sum = 0; i = 0; p1--;
cs2:    c = 1; j = 0;
cs3:    return;
}
```

(d)

```
void Tin_optimized() {
cs4:    READ(IN, sample, 1); sum + = sample;
        DATA = sample; i++;
cs5:    d = DATA;
        if (j == N) {
            j = 0; d = d * c;
            WRITE(OUT, d, 1);
        } else j++;
        if (p1 == 0) goto cs6;
cs7:    sum = 0; i = 0; p1--;
cs6:    if (i < N) return;
        DATA = sum / N; p1++;
        goto cs5;
}
```

(e)

Fig. 17. (a)–(c) Synthesized code for the initialization and reactive parts. (d), (e) Code after compiler optimizations.

Knowing that function `Init` is called only once before all other functions have a chance to run, the code for the initialization part can be optimized using constant propagation and dead code elimination: variable i is always 0, therefore the condition in the first line of cs_3 is always true if $N > 0$, and the `else` part can be eliminated. Also, variable p_5 is equal to 1 when the `switch` statement in cs_1 is executed, therefore the code can be further optimized. The new code is shown as function `Init_optimized` in Fig. 17(d). Function `Tin` can also be simplified, mainly by rearranging code and eliminating some useless `goto`s. The result is shown in Fig. 17(e).

The synthesized functions shown in Fig. 17 must be compared to those listed in Fig. 2(b), which were manually derived by looking at the specification. They are remarkably similar, the most important difference being that duplicated code in function `Tin` has disappeared in the automatically synthesized one, which uses variable p_1 to implement the same behavior.

V. Experimental Results

We used as our test system a moving pictures experts group (MPEG)-2 video decoder developed by Philips (see [21]). The system is composed of a set of concurrent processes, as shown in Fig. 18. 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 (i.e., forward and backward predictions) 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, which have FIFO semantics and can handle arbitrary data types. Philips used approximately 7700 lines of code to describe 11 processes, and 51 channels to connect them. An average of 16 communication primitives per process are used to transfer data through those channels.

In the original implementation, all processes were scheduled at run time using the plug-in Silicon Operating System (pSOS) RTOS. Our objective was to reduce run-time scheduling overhead due to context switchings, by merging processes as much as possible into quasi-statically scheduled ones. This also leads to further improvements in performance, since internal communication between merged processes reduces to simple assignments rather than a full FIFO implementation (e.g., as a circular buffer in memory).

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. Even though we generated PNs for other processes as well, we did not schedule the entire system, because we wanted to preserve some concurrence between processes and to verify the interaction between the generated code and the rest of the specification. Moreover, we report profiling results on a single processor machine, but this partition would also allow us to map the MPEG-2 video decoder to different tasks on multiple processors.The inputs to these five processes from the rest of the system are correlated. This means that once a reaction is triggered by the first input, the other will follow in a known sequence. We thus modeled all inputs as controllable, except the first one received, which is modeled as an uncontrollable source. As a result, our procedure generated a single source schedule for this trigger input, rather than several schedules for each of the original inputs.

The PN generated from the FlowC specification had 18 free choices, due to data-dependent conditions in the code; however, it was identified from the specification that ten of those free choices modeled conditions that were correlated, and thus the

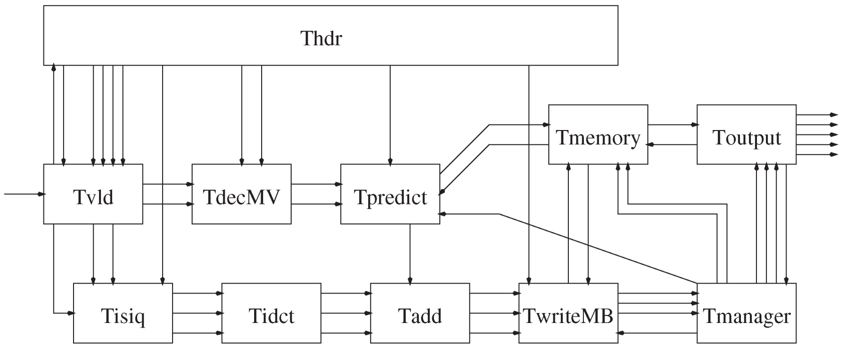
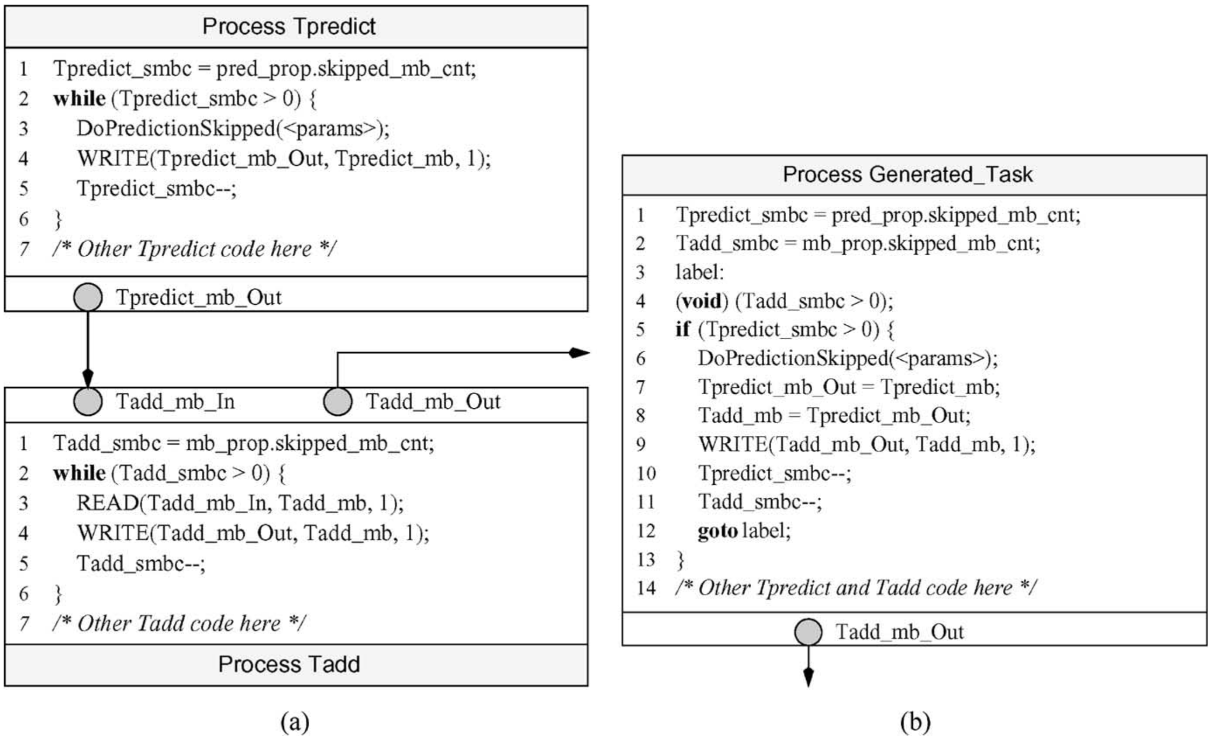


Fig. 18. MPEG-2 video-decoder block diagram.



```
Tpredict_smbc = pred_prop.skipped_mb_cnt;
while (Tpredict_smbc > 0) {
   DoPredictionSkipped(<params>);
   WRITE(Tpredict_mb_Out, Tpredict_mb, 1);
   Tpredict_smbc--;
}
/* Other Tpredict code here */
```

```
Tadd_smbc = mb_prop.skipped_mb_cnt;
while (Tadd_smbc > 0) {
   READ(Tadd_mb_In, Tadd_mb, 1);
   WRITE(Tadd_mb_Out, Tadd_mb, 1);
   Tadd_smbc--;
}
/* Other Tadd code here */
```

```
Tpredict_smbc = pred_prop.skipped_mb_cnt;
Tadd_smbc = mb_prop.skipped_mb_cnt;
label:
(void) (Tadd_smbc > 0);
if (Tpredict_smbc > 0) {
   DoPredictionSkipped(<params>);
   Tpredict_mb_Out = Tpredict_mb;
   Tadd_mb = Tpredict_mb_Out;
   WRITE(Tadd_mb_Out, Tadd_mb, 1);
   Tpredict_smbc--;
   Tadd_smbc--;
   goto label;
}
/* Other Tpredict and Tadd code here */
```

Fig. 19. (a) Example of FlowC specification. (b) Portion of the generated code for the MPEG-2 decoder.

technique described in [1] was used to model the correlation in the PN originally generated from the FlowC code. The resulting PN had 115 places, 106 transitions, and 309 arcs. Our algorithm generated a single process with the same interface as the original ones, which could be plugged into the MPEG-2 netlist, replacing the original five processes.

The data-dependent conditions modeled by free choices are resolved based on data values of the received video bit stream and local variables. Communication operations occur, depending on the outcome of these conditions. Also, the number of times a given communication is performed may also depend on them, for instance, in loops whose bounds are known only at run time, as shown in Fig. 19(a). The figure shows a small fragment of code taken from the two processes, `Tpredict` and `Tadd`, connected through a channel. Both processes implement `while` loops in which they exchange data: a macroblock is written from `Tpredict` to `Tadd`. A free choice is used to model the evaluation of the condition for each `while` loop. The source code is manually annotated to identify the correlation of the two choices.

On the other hand, Fig. 19(b) shows the same fragment of code automatically generated by our procedure, where the two were merged into a single entity. We generate a single loop, containing the appropriate interleaving of statements from the two original processes, similar to what an experienced designer would have written for a monolithic implementation. Note that the `WRITE` and `READ` 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` 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 an FIFO).

Of the two choices found in the processes' code of Fig. 19(a), matching to the conditions in the while loops, only one is used

```
...
if (!init_done) {
    READ(Tisiq_prop_seq_In, seq, 1);
    WRITE(Tidct_prop_seq_Out, seq, 1);
    init_done = TRUE;
}
...
```

Fig. 20. Run-time-dependent communication.

TABLE I
EFFICIENCY OF QSS HEURISTICS

QSS Method	Reachability Space			CPU
	Created	Final	Deleted	
All heuristics ON	902	902	0	10 s
t-invariants OFF	7842	1812	6030	30 s
All heuristics OFF	> 1500000	-	-	> 6000 s

as the termination condition of the merged loop in the generated code. This is because the two conditions were correlated, and the correlation was specified as described above. The other condition appears in line 4. The resulting code executes this statement because the expression may involve some operations other than the conditional check itself, but does not use the outcome of the condition.

The same situation was also observed in the code from process `Tisiq` shown in Fig. 20. A very similar piece of code is present in the receiving process `Tidct`, and the two are merged as a single entity in the schedule generated by our procedure, and thus in the resulting code.

We first show the efficiency of the heuristics used in the proposed scheduling algorithm. In Table I, columns "Created," "Final," and "Deleted" show the number of nodes visited during the scheduling, the number of nodes of the resulting schedule, and the difference between the two, i.e., the redundancy of a scheduling algorithm. The column "CPU" shows the CPU time needed to obtain a schedule, on a Sun Ultra Enterprise 250 with 2 GB of main memory and two processors running at 400 MHz. The first row shows the case where we used the t-invariant heuristic, as well as the heuristic with speculative-FCS described in the beginning of Section IV-B. It shows that the algorithm finds a schedule without visiting any additional node during the schedule search. In the second row, we disabled the t-invariant heuristic, while the other heuristic was still applied. The result shows that the search visited nearly nine times more nodes than the first case, deleted about 75% of them, and ended up with approximately twice as much generated code as the first case. Finally, when we also disabled the speculative-FCS heuristic, the search could not complete in reasonable time and memory: after 100 minutes, it had created more than 1 500 000 nodes and the memory size was around 400 MB.

Even for this size of PN obtained for the MPEG example, and even for the minimum bounds set on the places (equal to the maximum rates specified in the communication operations for the port modeled by each place), the reachability space of the PN includes several millions of states. It leaves little hope to successfully apply the exhaustive analysis of the space needed by the method in [17] when a schedule for minimally sized buffers does not exist. Contrary to [17], our approach generates assumptions on the required buffer size on the fly and manages to find schedules by exploring only a small part of the reachability space, as shown in Table I.

TABLE II
CPU TIME (SECONDS) OF THE MPEG-2 EXAMPLE

	Total	MPEG2			Test Bench	OS
		Total	Parser	5 Processes		
Original	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 III
CPU TIME (SECONDS) AND CODE SIZE OF THE FIVE SELECTED PROCESSES

	5 Processes				
	Total	Computation	Internal Communication	External Communication	Code Size
Original	3.72	1.01	2.23	0.48	18 K
QSS	1.57	0.96	0.13	0.48	24 K

Secondly, we compared the performance of the original concurrent specification of the MPEG-2 decoder with that of the same system where a single statically scheduled process is used in place of the five initial ones. 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 standard video stream composed of four images (one intra, one predicted, two bidirectional predicted).

Table II summarizes the total execution time, on the same Sun machine used for finding the schedule, for the two implementations. It also shows the individual contributions due to the processes implementing the MPEG-2 decoder (split among the parser and the five processes that we scheduled together), the test-bench, and the operating system (that dynamically schedules the tasks). 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.

Table III compares the execution times due to computation and communication of the five considered processes, both in the original system and in the quasi-statically scheduled one. As expected, computation and external communication (i.e., with the environment) are not significantly affected by our procedure. However, internal communication is significantly improved: this is because after scheduling it can be statically determined that all channels connecting the five considered processes never contain more than one data item at a time. Therefore, communication is automatically performed by assignment, rather than by using an FIFO or a circular buffer. The table also reports the object code size, which increases in the generated single task with respect to the five separated process, due to the presence of the control structures representing the static schedule in the synthesized code.

VI. CONCLUSION

This paper has proposed a method that bridges the gap between the specification and the implementation of reactive systems. From a specification given in terms of communicating processes, and by deriving an intermediate representation based on PNs, a set of concurrent tasks that serve input events with minimum communication effort is obtained.

We also presented a first effort towards automating this step. Experiments show promising results and encourage further research in the area.

We are currently working towards a more general definition of schedule, considering sequential and concurrent implementations on several resources (e.g., CPUs and/or custom datapaths) [6]. We are also planning to look further into providing a structural characterization of schedulability, if possible, for different classes of PNs. Another body of future research concerns an extension of the notion of a schedule into the time domain. Currently, timing guarantees come from the assumption that an environment is slow, while an implemented system is fast. If this is the case, then any event from the environment is served by a schedule that is bounded in length and space, i.e., has finite buffer sizes. However, such timing guarantees are weak when an implementation needs to meet predefined hard timing constraints on its response time. For timing critical applications, we work on extending our scheduling framework through explicit annotation of system events with delays and using timing-driven algorithms for a schedule construction.

REFERENCES

[1] G. Arrigoni, L. Duchini, L. Lavagno, C. Passerone, and Y. Watanabe, "False path elimination in quasi-static scheduling," in *Proc. Design Automation and Test Europe Conf.*, Paris, France, Mar. 2002, pp. 964–970.

[2] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings, "Fixed priority pre-emptive scheduling: An historical perspective," *Real-Time Syst.*, vol. 8, no. 2/3, pp. 173–198, 1995.

[3] J. Buck, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, 1993.

[4] J. T. Buck, "Static scheduling and code generation from dynamic dataflow graphs with integer valued control streams," in *Proc. 28th Asilomar Conf. Signals, Systems, and Computers*, Pacific Grove, CA, Oct. 1994, pp. 508–513.

[5] 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 *Proc. 37th Design Automation Conf.*, Los Angeles, CA, Jun. 2000, pp. 489– 494.

[6] J. Cortadella, A. Kondratyev, L. Lavagno, A. Taubin, and Y. Watanabe, "Quasi-static scheduling for concurrent architectures," *Fundam. Inform.*, vol. 62, no. 2, pp. 171–196, 2004.

[7] 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 signal processing systems," in *Proc. 37th Design Automation Conf.*, Los Angeles, CA, Jun. 2000, pp. 402– 405.

[8] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Norwell, MA: Kluwer, 1993.

[9] D. Har'el, H. Lachover, A. Naamad, A. Pnueli *et al.*, "STATEMATE: A working environment for the development of complex reactive systems," *IEEE Trans. Softw. Eng.*, vol. 16, no. 4, pp. 403–414, Apr. 1990.

[10] C. A. R. Hoare, *Communicating Sequential Processes*, ser. International Series in Computer Science. Englewood Cliffs, NJ: Prentice-Hall, 1985.

[11] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. Int. Federation Information Processing (IFIP) Congr.*, Stockholm, Sweden, Aug. 1974, pp. 471–475.

[12] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow graphs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.

[13] B. Lin, "Software synthesis of process-based concurrent programs," in *Proc. 35th ACM/IEEE Design Automation Conf.*, San Francisco, CA, Jun. 1998, pp. 502–505.

[14] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. Assoc. Comput. Mach.*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[15] H. Mathony, "Universal logic design algorithm and its application to the synthesis of two-level switching circuits," *Inst. Elect. Eng. Proc.*, vol. 136, pt. E, no. 3, pp. 171–177, May 1989.

[16] T. Murata, "Petri Nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[17] T. M. Parks, "Bounded scheduling of process networks," Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, Tech. Rep. UCB/ERL 95/105, Dec. 1995.

[18] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli, "Synthesis of embedded software using free-choice Petri nets," in *Proc. 36th ACM/IEEE Design Automation Conf.*, New Orleans, LA, Jun. 1999, pp. 805–810.

[19] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst *et al.*, "Scheduling hardware/software systems using symbolic techniques," in *Proc. Int. Workshop Hardware/Software Codesign*, Rome, Italy, 1999, pp. 173–177.

[20] F. Thoen, M. Cornero, G. Goossens, and H. De Man, "Real-time multitasking in software synthesis for information processing systems," in *Proc. Int. System Synthesis Symp.*, Cannes, France, 1995, pp. 48–53.

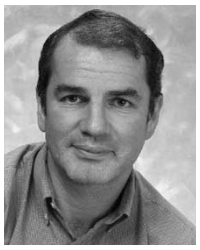
[21] P. van der Wolf, P. Lieverse, M. Goel, D. L. Hei, and K. Vissers, "An MPEG-2 decoder case study as a driver for a system level design methodology," in *Proc. 7th Int. Workshop Hardware/Software Codesign*, Rome, Italy, May 1999, pp. 33–37.

[22] P. Wauters, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-dynamic dataflow," in *Proc. 4th EUROMICRO Workshop Parallel and Distributed Processing*, Braga, Portugal, Jan. 1996, pp. 319–326.

Jordi Cortadella (S'87–M'88) received the M.S. and Ph.D. degrees in computer science from the Universitat Politècnica de Catalunya, Barcelona, Spain, in 1985 and 1987, respectively.

He is a Professor in the Department of Software at the Universitat Politècnica de Catalunya. In 1988, he was a Visiting Scholar at the University of California, Berkeley. His research interests include formal methods and computer-aided design of VLSI systems, with special emphasis on asynchronous circuits, concurrent systems, and logic synthesis. He has coauthored over 130 research papers in technical journals and conferences.

Dr. Cortadella has served on the technical committees of several international conferences in the field of design automation and concurrent systems. He received the Best Paper Award at both the International Symposium on Advanced Research in Asynchronous Circuits and Systems (2004) and the Design Automation Conference (2004).



Alex Kondratyev (M'94–SM'97) received the M.S. and Ph.D. degrees in computer science from the Electrotechnical University of St. Petersburg, St Petersburg, Russia, in 1983 and 1987, respectively.

He joined the R&D Coop TRASSA, St. Petersburg, Russia, in 1988, where he was a Senior Researcher. From 1993 to 1999, he was an Associate Professor in the Hardware Department at the University of Aizu, Fukushima-ken, Japan. In 2000, he joined Theseus Logic, Orlando, FL, as a Senior Scientist. Since 2001, he has been with the Cadence Berkeley Laboratories, Berkeley, CA, as a Research Scientist. He has coauthored two books on formal methods for asynchronous design, and has published over 70 journal and conference papers. His research interests include formal methods in system design, synthesis of asynchronous circuits, computer-aided-design methodology, and theory of concurrence.

Dr. Kondratyev was a cochair of the Async'96 Symposium, cochair of the CSD'98 Conference, and has served as a member of the program committees for several conferences.

Luciano Lavagno (S'88–M'93) graduated *magna cum laude* in electrical engineering from the Politecnico di Torino, Turin, Italy, in 1983. In 1992, he received the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley.

He was with CSELT Laboratories, Turin, Italy, from 1984 to 1988. In 1988, he joined the Department of Electrical Engineering and Computer Science, University of California, Berkeley, where he worked on logic synthesis and testing of synchronous and asynchronous circuits. Between 1993–1998, he was an Assistant Professor at the Politecnico di Torino and between 1998–2001, he was an Associate Professor with the University of Udine, Udine, Italy. Between 1993–2000, he was the Architect of the POLIS project (a cooperation between the University of California, Berkeley, Cadence Design Systems, Magneti Marelli, and Politecnico di Torino), developing a complete hardware/software codesign environment for control-dominated embedded systems. He is currently an Associate Professor with the Politecnico di Torino, and a Research Scientist with Cadence Berkeley Laboratories, Berkeley, CA. He is a coauthor of two books on asynchronous circuit design and a book on hardware/software codesign of embedded systems, and has published over 100 journal and conference papers. His research interests include the synthesis of asynchronous and low-power circuits, the concurrent design of mixed hardware and software embedded systems, and dynamically reconfigurable processors.

Dr. Lavagno received the Best Paper award at the 28th Design Automation Conference in 1991, in San Francisco, CA. He has served on the technical committees of several international conferences in his field (e.g., the Design Automation Conference, the International Conference on Computer-Aided Design, the International Conference on Computer Design, and Design Automation and Test in Europe) and of various other workshops and symposia.

Claudio Passerone (M'98) received the M.S. degree in electrical engineering from the Politecnico di Torino, Turin, Italy, and the Ph.D. degree in electrical engineering and communication from the same university, in 1994 and in 1998, respectively.

He is currently an Assistant Professor in the Electronic Department of Politecnico di Torino. His research interests include system-level design of embedded systems, electronic system simulation and synthesis, and reconfigurable computing. He is a coauthor of a book on hardware/software codesign of embedded systems, and has published over 40 journal and conference papers.

Dr. Passerone has served on the technical committee of the Design Automation and Test in Europe Conference. In 2002, he received the Best Paper Award at the 9th International Conference on Electronics, Circuits, and Systems.

Yosinori Watanabe (S'88–M'93) received the Ph.D. degree in electrical engineering and computer sciences from the University of California, Berkeley, in 1994.

He joined Digital Equipment Corporation, Maynard, MA, in 1994. He was a Member of the Design Team for the ALPHA microprocessor, while being engaged in logic synthesis for high-performance microprocessors. Since 1997, he has been with Cadence Berkeley Laboratories, Berkeley, CA, where he has been involved in research projects for developing a design environment and methodologies for embedded systems.

Dr. Watanabe received the IEEE Circuits and Systems (CAS) Society Outstanding Young Author Award and the IEEE CAS Best Paper Award from the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS in 1995 and 1998, respectively.