Task Generation and Compile-Time Scheduling for Mixed Data-Control Embedded Software *

Jordi Cortadella
Universitat Politècnica de Catalunya

Alex Kondratyev
University of Aizu

Luciano Lavagno
Università di Udine

Marc Massot
Universitat de Girona

Sandra Moral
Universitat Politècnica de Catalunya

Claudio Passerone
Politecnico di Torino

Yosinori Watanabe
Cadence European Laboratories

Alberto Sangiovanni-Vincentelli
University of California at Berkeley

Abstract

A method for synthesizing code for the software component of a system is proposed. The specification is given as a set of concurrent processes that communicate through channels. Each process is a sequential program that may contain data-dependent control statements.

The synthesized software consists of a set of tasks. A task is generated by analyzing the computation associated to the occurrence of an event at each input port connected to the environment. Our task generation and scheduling algorithm guarantees that task execution can be performed with a finite amount of inter-task buffer memory under arbitrary input streams.

Petri nets are used as the underlying model to formally analyze our algorithms. This model is also used to derive several algorithms for optimizing code generation for the set of tasks yielding compact and high-performance software implementations.

*This work has been supported by the European Comission within the ESPRIT/OMI COSY project EP25443.

1 Introduction

The Problem We consider a system to be specified as a set of concurrent processes. A set of input and output ports are defined for each process, and point-to-point communication between processes occurs through uni-directional channels between ports. We support multi-rate communication, in which the number of objects read or written by a process at any given time may be an arbitrary constant. A process may communicate with the environment in which the system is executed. This is done through input and output ports for which no channel is defined. Such primary input ports can belong to one of two classes, which we call controllable and uncontrollable. The latter is used to trigger an execution of the system, i.e. when the system receives an object at an uncontrollable port, it reacts to the environment by performing operations. On the other hand, the system may request the environment for further inputs through controllable ports, while this is not allowed for uncontrollable ports. We assume that the class of each input port is defined in the functional specification. Output ports are always written under system control, and the environment must be ready to accept them at any time (as allowed by the concurrent process specification). We restrict our attention to processes described as sequential programs and whose implementation is mapped as software to be executed on a programmable processor. The sequential program for each process is specified in a language based on C, extended in order to specify communication operations, and may contain both arithmetic operations and data-dependent control statements (e.g., `if-then-else` or arbitrary `while` or `for` loops).

This concurrent specification mechanism permits the underlying implementation architecture (number of processors, scheduling policy, implementation of communication, HW/SW partitioning, etc.) to be varied for a given functional specification, thus requiring a much reduced re-design effort with respect to more traditional methods in which the tasks for each processor are explicitly specified since the beginning [5].

We address the problem of generating highly efficient code (software synthesis) from this concurrent specification. The synthesis process consists of 1) defining real-time tasks from functional processes, a non-trivial problem since mapping each process to a separate software task could be very inefficient, due to high inter-task communication and context switching costs [5], and 2) sequentializing the code of the processes for each task so that running time and code size is optimized. The complete software implementation of the system includes real-time scheduling of the tasks that we assume to be managed by an embedded operating system and is outside the scope of this paper.

The Algorithm The algorithm described in this paper first generates a task for each input uncontrollable port, and then associates a sequential program with the task. The sequential program is called a *schedule* for the input port, and has the property that it can be executed with finite memory for inter-task lossless channels, assuming arbitrary input streams. A schedule can be obtained by first finding operations specified in various concurrent processes that need to be executed when this port receives an input from the environment, and sequentializing them, while ensuring their execution with finite memory for the communication channels. The resulting schedule, if one is found, determines an upper bound on the quantity of objects stored at a time for each channel during the execution. If an upper bound for a channel is given in the specification, a schedule that guarantees the execution within the bound is sought. The specification may contain data-dependent control constructs, and thus a total order of these operations cannot be determined in general until run-time, when values of data at the constructs become known. Therefore, the operations are sequentialized to reduce run-time overhead maximally, with which only resolutions of the control constructs are made at run-time.

We use a class of Petri nets as the underlying model, since it can represent data-dependent control and concurrency explicitly in the structure of a net. The specification is translated into a single Petri net, and a set

of schedules is computed, first as directed graphs annotated with objects of the Petri net. Each schedule is then transformed into a software program by traversing the corresponding graph. An important issue in generating a set of schedules is interference among them, i.e. an execution of one schedule may disable another schedule to be executed further. This may happen, for example, when the same channel is accessed by two schedules and objects at the channel that one schedule needs to read for continuing its execution has been read by the other schedule. We introduce notion of *independence* among schedules to avoid the interference, and show that the proposed algorithm guarantees the independence of the resulting set of schedules.

We have implemented the entire software synthesis flow from specifications to synthesized tasks in a set of tools, which comprise compiler, linker, scheduler and code generator, and applied it to an industrial multi-media example to show effectiveness of our approach.

Related Work The scheduling problem has been the subject of significant research, especially for specifications modeled by variations of Dataflow networks, such as Static (or Synchronous) and Boolean Dataflow (SDF and BDF) [2, 3]. SDF specifications can be statically scheduled with a variety of cost functions, for single and multiple processors, but make the limiting assumption that there is no data-dependent control construct. Although this assumption may be acceptable for some applications, it is increasingly difficult to satisfy in modern embedded systems. BDF, on the other hand, can model such constructs, but the problem of determining the existence and deriving a finite-memory schedule for BDF in general is undecidable [3]. Approaches that use variations of control-data flow graphs, proposed mainly in the context of high-level synthesis for hardware design [7, 1], also allow both control and data operations in functional specifications. However, they cannot explicitly model the communication semantics often used in embedded systems, such as multi-rate data communication, and thus are applicable only to a limited class of applications for software design. The same limitation applies to the software synthesis techniques proposed in [8] and [14], which can be applied only to closed systems with single-rate communication. The work of [12] is related to ours, especially in the underlying Petri net model, but cannot handle synchronization-dependent choice, nor multiple reads/writes from/to the same channel by a given process. Finally, the authors of [13] also use a Petri net-like representation and can handle data-dependent and synchronization-dependent choice, but they require the designer to explicitly specify bounds on the maximum size of each communication channel, while we can handle user-specified bounds as well as determine the size of unbounded channels (see Section 4.4 for a further comparison with this approach).

Organization After presenting in Section 2 terminology on Petri nets, we present in Section 3 the language to specify a system function as a network of processes and the procedures to translate it into a Petri net. Section 4 formally defines schedules and provides theoretical validation of the proposed approach. In Section 5, we propose an algorithm for finding schedules. The algorithm can use various termination conditions, and we show that it can always find a schedule if one exists under the given termination condition. Section 6 presents a procedure to transform the schedule to a set of software tasks, each specified as a sequential program. In Section 7 we analyze the implications of synchronization-dependent choice and the problem of false paths that can arise in modeling with Petri nets and discuss possible solutions. In Section 8 we give experimental results obtained running the algorithm on an industrial video application provided by our partners.

2 Petri Nets

A Petri net (PN) is defined by a tuple (P, T, F, M_0), where P and T are sets of *places* and *transitions* respectively. F is a function from $(P \times T) \cup (T \times P)$ to non-negative integers. A *marking* M is a function

from P to non-negative integers, where its output for a place p is denoted by $M[p]$, which we call the number of *tokens* at p in M. If $M[p]$ is positive, the place p is said to be *marked* at M. M_0 is a marking, which we refer to as the *initial marking*. A Petri net can be represented by a directed bipartite graph, where an edge $[u, v]$ exists if $F(u, v)$ is positive, which is called the *weight* of the edge. We may call v a *successor* of u and u a *predecessor* of v, respectively. A transition t is *enabled* at a marking M, if $M[p] \geq F(p, t)$ for all p of P. In this case, one may *fire* the transition at the marking, which yields a marking M' given by $M'[p] = M[p] - F(p, t) + F(t, p)$ for each p of P. In the sequel, $M \xrightarrow{t} M'$ denotes the fact that a transition t is enabled at a marking M and M' is obtained by firing t at M. A sequence of transitions $(t_1, \ldots, t_k)$ is said to be *fireable* from a marking M, if there exists a sequence of markings $(M_1, \ldots, M_{k+1})$ such that $M_1 = M$ and $M_i \xrightarrow{t_i} M_{i+1}$ holds for each $i = 1, \ldots, k$. A transition t is said to be a *source*, if $F(p, t) = 0$ for all p of P.

A marking M' is said to be *reachable* from M if there is a sequence of transitions fireable from M that leads to M'. The set of markings reachable from the initial marking is denoted by $\mathcal{R}(M_0)$. The *reachability graph* of a Petri net is a directed graph in which $\mathcal{R}(M_0)$ is the set of nodes and each edge $[M, M']$ is a transition t with $M \xrightarrow{t} M'$. The *reachability tree* of a Petri net is a tree in which each node is labeled with a marking of $\mathcal{R}(M_0)$, the root node is labeled with M_0, and each edge $[v, v']$ represents a transition t with $M \xrightarrow{t} M'$, where M and M' are the labels of v and v'. Each path starting at the root of the reachability tree represents a sequence of transitions fireable from M_0.

A key notion we use in Petri nets for defining schedules is *equal conflict sets*. A pair of non-source transitions t_i and t_j is said to be in *equal conflict*, if $F(p, t_i) = F(p, t_j)$ for all p of P. These transitions are in conflict in the sense that t_i is enabled at a given marking if and only if t_j is enabled, i.e. if the firing of one transition disables t_i, it also disables t_j. The equal conflict is an equivalence relation defined on the set of non-source transitions, and each equivalence class is called *equal conflict set* (ECS). As a special case, we also define as an ECS a set that consists of a single source transition. By definition, if one transition of an ECS is enabled at a given marking, all the other transitions of the ECS are also enabled. Thus, we may say that this ECS is *enabled* at the marking.

A place p is said to be a *choice* place if it has more than one successor transition. A choice place is *Equal Choice* (a generalization of free choice [11]) if all the successor transitions are in the same ECS. A Petri net is *Equal Choice* if all choice places are equal. A choice place is *unique* if no more than one successor transition can be enabled in any of the markings of $\mathcal{R}(M_0)$. A *unique-choice Petri net* (UCPN) is that in which all choice places are either equal or unique.

3 Specification of System Functions

A system function is represented as a network of processes. A process is specified as a sequential program written in *FlowC*, a language based on C and enhanced with primitives to receive/send data from/to ports. A process has a set of input and output ports to communicate with other processes or with the environment. The network of processes is finally built by specifying a set of communication channels. Each channel communicates two processes by connecting an output port of one process to an input port of another process. Unconnected ports are assumed to be connected to the environment.

The operations to communicate through ports have the following syntax:

```
READ_DATA (port, data, nitems)
WRITE_DATA (port, data, nitems).
```

The parameter `nitems` denotes the number of logical data units involved in the transfer. For example, the producer of an image may transfer a line of pixels in one port operation such as `WRITE_DATA(p,line,625)`, where `line` denotes a buffer storing 625 pixels. The consumer may read the line in a pixel-by-pixel basis by executing a loop with the operation `READ_DATA(p,pixel,1)` 625 times.

Operations with ports have *blocking semantics*. A `read` blocks when the number of items in the channel is smaller than `nitems`. Similarly, a `write` blocks, if a bound is pre-defined at the channel and the number of items in the channel would exceed this bound after writing.

The network of processes is transformed into a single Petri net, which is built in two steps. In the first step, called *compilation*, a Petri net for each process is constructed and each port is associated to a place of the Petri net. The second step, called *linking*, builds a Petri net by "connecting" the Petri nets according to the defined channels. Next, both steps are described in more detail.

3.1 Compilation

A specification in *FlowC* is translated into a set of Petri nets, one for each process, that communicate through ports represented by places. Each transition is annotated with a fragment of C code. Processes are sequential and, therefore, their corresponding Petri nets have no concurrency.

The compilation process attempts to generate the most compact Petri net that preserves the observable behavior at the level of ports. Thus, a `while` statement can be represented by only one transition if no port operations are executed in its body. On the other hand, the same statement will be represented by several transitions if some channel operation is present in its body. An example will be given in short.

Conditions at control flow statements are represented by *Equal Choice* places with the corresponding annotated boolean expression and two outgoing arcs labeled `True` and `False`. The successor transitions constitute an ECS.

The level of granularity at which each statement is represented is determined by the calculation of *leaders* in the code, according to the following rules:

1. The first statement of the process is a leader.
2. A `READ_DATA` statement is a leader.
3. Any statement which immediately follows a `WRITE_DATA` statement is a leader.
4. The first statement of a control flow statement (e.g. `while`, `if-else`, etc.) that contains a leader is a leader.
5. Any statement that immediately follows a control flow statement that contains a leader is a leader.

Every portion of code consists of a leader and all statements up to the next leader or the end of the process.

Figure 1 shows a process specification in *FlowC*. The process computes all divisors of the numbers read by port `in`. The greatest divisor is sent to port `max`, whereas all divisors are sent to port `all`. The leaders are the statements at lines 4 (by rules 2 and 4), 9 (by rule 3), 11 and 13 (by rule 4).

The construction of the Petri net for a process is done by traversing the parse tree in two steps. The first step (bottom-up) determines which statements have port statements in their body and calculates the leaders. The second step constructs the Petri by successive refinement of transitions. Initially, the Petri net has one marked place and one transition. Those transitions containing port statements must be refined. The refinement of port statements requires the creation of weighted arcs from/to the place representing the port. Figure 2 depicts two examples of refinement.

```
 1: PROCESS divisors (In_DPORT in, Out_DPORT max, Out_DPORT all) {
 2:     int n, i;
 3:     while (1) {
 4:         READ_DATA(in , &n, 1);
 5:         i = n/2;
 6:         while (n%i!=0)
 7:             i−−;
 8:         WRITE_DATA(max, i, 1);
 9:         WRITE_DATA(all, i, 1);
10:         while (i < 1) {
11:             i−−;
12:             if (n%i==0)
13:                 WRITE_DATA(all, i, 1);
14:         }
15:     }
16: }
```

Figure 1: A process specification in *FlowC*.

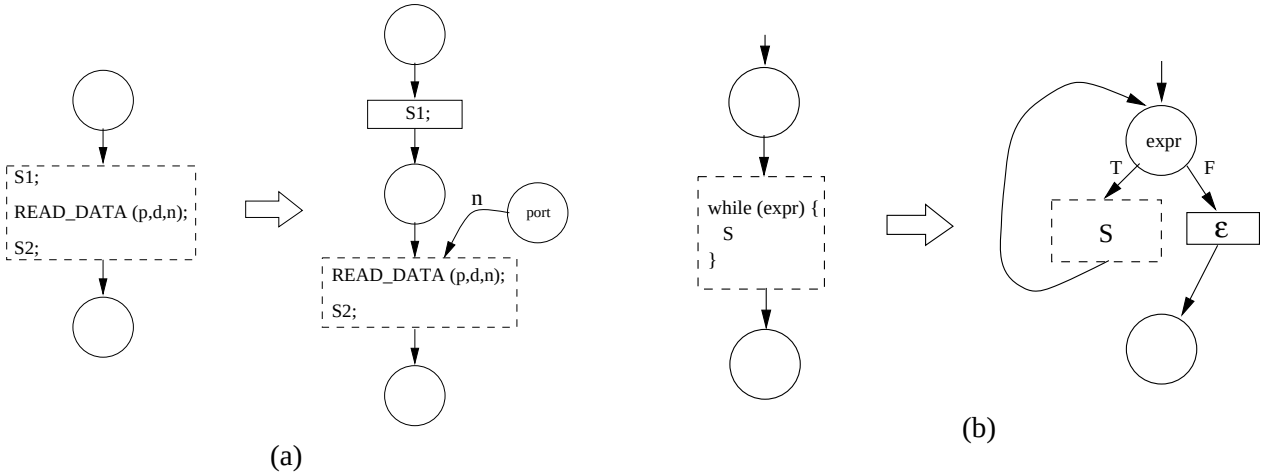


Figure 2: Refinement of transitions: (a) `READ_DATA` statement, (b) `while` statement (ε denotes a silent transition).

If we ignore the places associated to the ports, the Petri net of one process obtained by the compilation strategy mentioned above has the following properties:

- Exactly one place is marked at each reachable marking. The token mimics the “program counter” of the sequential process.
- It is Equal Choice.

When places associated to ports are also considered, new choice places may arise. This occurs when the same process reads data from the same port in different statements, thus the place representing the port is a choice. However, these places are unique choices, since no pair of “read” transitions can be simultaneously enabled competing for the same port. Thus the resulting Petri net is UCPN.

Figure 3 depicts the Petri net corresponding to the process specified in Figure 1.

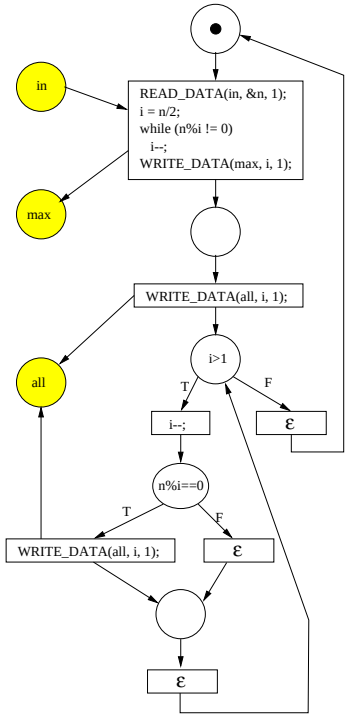


Figure 3: Petri net obtained from the specification of Figure 1

.

3.2 Linking

After compilation, a Petri net is obtained for each process. Each Petri net has some dangling places representing the ports.

Linking combines Petri nets generated in compilation into one, by merging each pair of places for ports connected by a channel. This ensures the resulting Petri net is still unique-choice. A semantic check is performed to ensure that the ports connected by a channel have the same data type. If a bound is defined for a channel, it is represented as an attribute for the merged place.

For an input (output) port connected to the environment, a source (sink) transition is connected to the place for the port, where the weight of the arc denotes a specified rate of the port. The control or uncontrol attribute is specified for each input port of this kind, and the corresponding source transition is called *controllable* or *uncontrollable* respectively.

4 Schedules

This section formally defines schedules and presents their requirements for correct execution.

4.1 Definition

A task is generated for each uncontrollable input port, and thus we compute a schedule for each uncontrollable source transition. A *schedule* for a given uncontrollable source transition a is a directed graph. A node v is associated with a marking denoted by $M(v)$, and an edge e is associated with a transition denoted by $T(e)$. The graph has a distinguished node r and has five properties. First, r is associated with the initial marking, and has an out-degree of 1. Second, the edge out of r is associated with a. Third, for each node v, the set of transitions associated with the edges out of v is an ECS enabled at $M(v)$. Fourth, for each edge $[u, v]$, $M(u) \stackrel{T([u,v])}{\rightarrow} M(v)$ holds. Fifth, each node is on at least one directed cycle that includes r.

Intuitively, scheduling can be deemed as a game between a scheduler and the environment. Starting from the node r, once the environment fires the transition a, the scheduler traverses the schedule for a, firing transitions of the visited edges. When it reaches a node whose out-going edge is associated with an uncontrollable source transition, it ceases the traversal, waiting for the environment to fire the transition. The scheduler resumes the traversal, as soon as the firing occurs. If it comes to a node with the out-degree greater than 1, one of the out-going edges is taken, and the traversal continues by firing the associated transition. At such a node, the ECS defined by the out-going edges has more than one element. The FlowC compiler introduces such an ECS to model a data-dependent control construct such as `if-then-else`, where each resolution of the control is modeled by a single transition. As the resolution of the control is determined not by the scheduler but by values of the data at the construct, a schedule must be made so that no matter which out-going edge is chosen at the node, the traversal can be continued. Furthermore, the fifth property guarantees that at any moment of the traversal, there is a path to return to r, which ensures cyclic behavior of the schedule [1].

A transition t is said to be *involved* in a schedule if there is an edge in the schedule with which t is associated. Similarly, a place p is said to be *involved* in a schedule if p is a predecessor of a transition involved in the schedule. A node of a schedule is said to be an *await node*, if its out-going edge is associated with an uncontrollable source transition. By definition, the distinguished node r is an await node. If the same uncontrollable source transition is associated for all the await nodes of a schedule, it is called a *single source schedule*, or an *SS schedule* for short. An SS schedule for an uncontrollable source transition a may be denoted by $SSS(a)$. Note that $SSS(a)$ may contain more than one await nodes, and that it may involve controllable source transitions. If a schedule is not an SS schedule, it is called a *multiple source schedule*, or an *MS schedule* for short. Figure 4(a) shows a PN with two uncontrollable source transitions and their corresponding SS schedules. On the other hand, the PN in Figure 4(b) has no SS schedules, if both a and b

[1] The initialization sequence of operations is not included in a schedule, as this paper considers scheduling of cyclic behavior executed repeatedly in response to the environment.

are uncontrollable source transitions, because the fifth property of the schedule definition requires firing both uncontrollable transitions a and b[2].

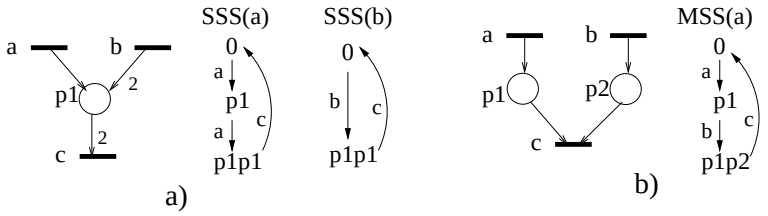


Figure 4: Single(a) and multiple (b) source schedules

4.2 Single source schedules

In the analogy of the game argument given in the previous section, for a given set of schedules, one for each uncontrollable source transition, when the environment produces a sequence σ of uncontrollable source transitions, the system responds to each symbol of σ by traversing a path of the schedule for the uncontrollable source transition represented by the symbol, until an await node is encountered. The sequence of paths traversed in this way characterizes the system behavior realized by the schedules for the input sequence σ. We call such a sequence of paths a *run* of the set of schedules.

Definition 4.1 *Given a set S of schedules and a finite sequence $\sigma = \sigma_1 \cdots \sigma_k$, where σ_i is an uncontrollable source transition, a run of S with respect to σ is an ordered set $(\pi_1, \ldots, \pi_k)$ with four properties:*

1. *For any i, π_i is a directed path between a pair of await nodes of the schedule for σ_i in S, with no await node contained in-between.*
2. *For any i, the transition associated with the first edge of π_i is σ_i.*
3. *For any uncontrollable source transition a, if σ_i is the first occurrence in σ such that $\sigma_i = a$, then the first node of π_i is the distinguished node of the schedule for a.*
4. *For any uncontrollable source transition a and for any two symbols σ_i and σ_j, if $\sigma_i = \sigma_j = a$, $i < j$, and no symbol between σ_i and σ_j is equal to a, then the last node of π_i is equal to the first node of π_j.*

The first property above says that a traversal of a schedule is made between a pair of await nodes at a time. The second one means that the i-th traversal begins with a firing of the uncontrollable source transition represented by the i-th symbol of σ. The third property says that the initial traversal of a schedule starts with the distinguished node of the schedule. The final property guarantees that when a schedule is traversed, the traversal starts with the await node reached at the end of the previous traversal of that schedule.

Note that a run of S with respect to σ is not unique in general, since a schedule may contain a node with the out-degree greater than 1, and thus more than one paths may exist from a given await node. We also note that a run may not always exist. For example, in the PN and the schedule for a shown in Figure 4-(b), no run exists for $\sigma = aa$. This is because, the first occurrence of a in σ causes a traversal from the distinguished node 0 to an await node p_1, and the traversal for the second occurrence of a needs to start with an edge

[2]The same PN would have SS schedules if either a or b was specified as *controllable*.

from p_1 to p_1p_2. However, the transition associated with this edge is b, and the second property above does not hold. This problem occurs because the schedule involves both of the uncontrollable source transitions a and b. Since the edge is associated with an uncontrollable source transition b, the only way to continue a traversal in the schedule is to fire b. However, this schedule is for the other uncontrollable transition a, and the system will never use this schedule to serve an occurrence of b. If, instead, we allowed the environment to keep producing a's, while the schedule for a is allowed to get stuck at the await node p_1, then it results in unbounded accumulation of tokens at the place p_1 in the original PN shown in Figure 4-(b). Hence, in order for a set of schedules to be executed, each of them has to be a single source schedule. Formally, the following holds from the definition of a run.

Proposition 4.1 *A set S of schedules has a run for any finite sequence of uncontrollable source transitions, if and only if S consists only of single source schedules.*

For the UCPN class, SS schedules have additional attractive properties, which are given in the rest of this subsection.

Property 4.1 *Given a UCPN, if a transition* b *is enabled in the initial marking* M_0 *and* b *is not involved in* SSS(a) *then any transition that has a path to* b *in the PN graph is not involved in* SSS(a) *as well.*

Proof: The proof is done by induction on the length of path between c and b.

1. Induction basis. Let transition c be a direct predecessor of b, i.e. a successor place of c is a predecessor place of b. The length of the path between c and b measured in the number of transitions is 1, and we say that c is backward reachable from b in one step. Let us consider an arbitrary place p which is a predecessor of b. From b being enabled in M_0, it follows that p must have tokens in M_0. None of them can be consumed by the firing of a transition, say d, involved in $SSS(a)$ because d must be in conflict with b and hence involving d in $SSS(a)$ implies involving b. Therefore the token count in any predecessor place of b remains the same in all nodes of $SSS(a)$. The latter clearly means that no transition c directly preceding b is involved in $SSS(a)$.

Induction step. Suppose that the property is valid for any transitions which are backward reachable from b in k steps. Let c be a backward reachable transition from b in $k+1$ steps. Then there exists a transition d which is backward reachable from b in k steps and which is 1-step forward reachable from c. By the induction assumption d is not involved in $SSS(a)$. Hence we can apply the considerations similar to those from the induction basis and arrive to the conclusion that c is also not involved in $SSS(a)$. ■

Property 4.1 tells us that SS schedules partition PN transitions into equivalence classes based on backward reachability, i.e. the absence of a transition from the schedule implies the absence of all its predecessors as well.

Property 4.2 *Given a UCPN, any transition* b *which is not enabled in the initial marking* M_0 *but is enabled in some of the nodes of* SSS(a) *is involved in* SSS(a).

Proof: The proof is trivial. If b is not enabled in M_0 then there exists a place p such that p is a predecessor of b and p does not have enough tokens at M_0 to enable b. If b becomes enabled in the marking of a node v in $SSS(a)$, then p has more tokens in $M(v)$ than in M_0. By the definition of a schedule, there is a path from v to the distinguished node r. Since $M(r) = M_0$, b should belong to this path to consume extra tokens from p at $M(v)$ (note that if some other transition c consumes p then due to ECS properties b and c are in equal conflict and both should be involved in $SSS(a)$). ■

Property 4.2 states the notion of fairness for SS schedules in UCPNs because every transition which becomes enabled in an SS schedule must be given a chance to fire within the schedule. The next property extends the fairness for initially enabled transitions.

Property 4.3 *Given a UCPN, if a transition is enabled in the initial marking but not involved in an SS schedule of any uncontrollable transition, there exists a schedule for the transition that does not involve uncontrollable transitions.*

Proof: The proof immediately follows from the Property 4.1 because if b is not involved in $SSS(a)$ then uncontrolled transition a is not in the transitive fan-in of b and hence a is required to refill the tokens which will be consumed by the firing of b. ■

Property 4.3 tells first that for any initially enabled transition there exists a corresponding schedule, and second, it states that if a transition is not involved in any schedule of uncontrollable transitions then this is fair in the other way around as well: a schedule which involves this transition does not involve uncontrollable transitions [3].

4.3 Interference of schedules

When an SS schedule for an uncontrollable transition a is derived, every occurrence of a made by the environment will be served by the system according to the schedule. However the environment may produce uncontrollable transitions in any order. Therefore during implementing a particular SS schedule the system could be forced to switch to another schedule to serve the occurrences of other uncontrollable transitions. This poses the problem of interference among schedules.

To illustrate this problem let us start from an example. Figure 5 shows a PN with two uncontrollable transitions a and d and their SS schedules. It is easy to see that these two schedules are non-interfering because whenever the environment produces an uncontrollable transition the system serves it and returns to the initial marking. For this example, given $SSS(a)$ and $SSS(d)$ the system can serve any order of uncontrollable transitions.

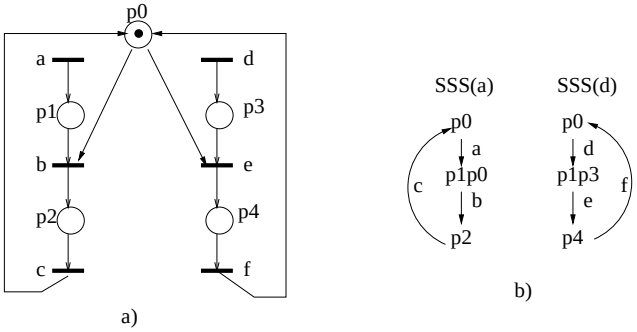


Figure 5: PN with non-interfering schedules

Unfortunately, the situation is more complex in general. Figure 6(a) shows a PN that is obtained from a PN in Figure 5(a) by changing weights on input and output edges of c and f. In the corresponding SS schedules (Figure 6(b)), it is not possible to return to the initial marking after every firing of a source transition because more than one await nodes exist in each schedule. Suppose that the environment produces a sequence $\sigma = ad$. Then a is served in accordance to $SSS(a)$ until an await node $p2p0$ of $SSS(a)$ is reached. The occurrence of d forces the system to switch to the schedule $SSS(d)$ and fire a transition e to reach an await

[3] Note that our procedure computes a schedule for each uncontrollable source transition only, and does not generate schedules for those transitions that are initially enabled but not involved in the schedules for uncontrollable sources.

node p_4p_0. The sequence of transitions fired so far by these traversals is $abde$. In the original PN of Figure 6-(a) representing the entire system, this sequence leads a marking to p_4p_2, as shown in Figure 6-(c). From this marking, it is impossible to reach a marking so as to enable transitions b and e, and thus a traversal of $SSS(a)$ or $SSS(d)$ cannot be continued further.

Two important observations come out from this example:

1) SS schedules may interfere with each other, i.e. firing transitions based on a traversal of one schedule may disable the other schedule to continue a further traversal.

2) Each SS schedule keeps track of the marking changes locally (looking only at transitions from its own schedule). When interference occurs, some other schedules might change a marking of the shared places, which makes the local marking (observable by a particular SS schedule) different from the marking of the entire PN (in the PN of Figure 6(a), while $SSS(d)$ assumes the "local" initial marking to be p_0p_0, in fact the firing of b in $SSS(a)$ has distorted it to p_2p_0).

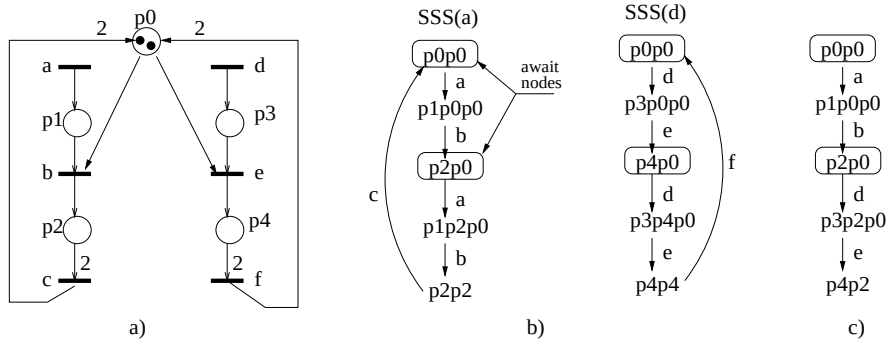


Figure 6: SS schedules with interference. Circled nodes in (b) are await nodes.

To treat this problem formally, we first define the executability of a set of schedules.

Definition 4.2 *For a given a PN, a set of SS schedules is said to be executable, if for any finite sequence σ of uncontrollable source transitions, and for any run π of S with respect to σ, the sequence of transitions defined by π is fireable from the initial marking of the PN.*

The sequence of transitions defined by a run $\pi = (\pi_1, \ldots, \pi_k)$ designates the sequence $seq_1 \cdots seq_k$, where seq_i is the sequence of transitions associated with the path π_i for each i. If a set of SS schedules is executable, for any sequence of uncontrollable source transitions given by the environment, the system can respond to traverse the schedules, and for any of such runs obtained by the traversals, the sequence of transitions associated with the paths of the run can be fired in the original PN from the initial marking.

A question is then how to check whether a given set of schedules is executable. A naive approach of simulating the PN with sequences of transitions defined by different runs of the schedules is not practical, and we are interested in checking the executability statically by analyzing the graphs of SS schedules. We therefore introduce notion of schedule *independence*.

Definition 4.3 *Two SS schedules are said to be mutually independent iff: for any place p involved in one schedule, $M(v)[p]$ is a constant over all the await nodes v of the other schedule.*

A set of SS schedules is said to be independent if each pair of schedules in it is mutually independent.

Intuitively, independence means that for each place p used by one schedule (i.e., such that the token count of p changes during the execution of the schedule), the other schedule has the same number of tokens at p at all its await nodes. We claim that an independent set of SS schedules is executable.

Proposition 4.2 *If a set S of SS schedules (one for each uncontrollable source transition of a given PN) is independent, S is executable.*

Proof: For a given run $\pi = (\pi_1, \ldots, \pi_k)$, let M_l be the marking of the PN reached by firing the sequence of transitions defined by $(\pi_1, \ldots, \pi_l)$ from the initial marking M_0, where $l = 1, \ldots, k$. If the sequence is not fireable from M_0, then M_l is said to be undefined. For the schedule $SSS(a_i)$ of S for an uncontrollable source transition a_i, let $await(SSS(a_i), l)$ denote the await node at which $SSS(a_i)$ resides at the end of the traversal of π up to π_l. In particular, let $await(SSS(a_i), 0)$ denote the distinguished node of $SSS(a_i)$.

We show that if S is independent, for any sequence $\sigma = \sigma_1 \cdots \sigma_k$ of uncontrollable source transitions, and for any run π of S with respect to σ, the marking associated with the await node $await(SSS(a_i), l)$ and the marking M_l of the PN coincide at all the places involved in $SSS(a_i)$, i.e. for any a_i and any place p involved in $SSS(a_i)$, $M(await(SSS(a_i), l))[p] = M_l[p]$ for $l = 1, \ldots, k$. Note that this property implies that S is executable.

Suppose for the sake of contradiction that a schedule $SSS(a_i)$, a place p involved in $SSS(a_i)$, and an index l exist for which this equality does not hold, i.e. either M_l is undefined or $M(await(SSS(a_i), l))[p] \neq M_l[p]$. We show that S is not independent in this case. Without loss of generality, suppose that l is the smallest index for which this violation arises. Namely, M_{l-1} is defined and $M(await(SSS(a), l-1))[p'] = M_{l-1}[p']$ holds for any schedule $SSS(a)$ of S and any place p' involved in $SSS(a)$. Then M_l is defined and $M(await(SSS(\sigma_l), l))[p'] = M_l[p']$ holds for any place p' involved in $SSS(\sigma_l)$. It follows that $\sigma_l \neq a_i$. Since $\sigma_l \neq a_i$, the await nodes at which the schedule for a_i resides before and after the traversal of π_l are the same, and thus $M_{l-1}[p] \neq M_l[p]$. This implies that there is a transition in π_l whose firing changes the token count at p, i.e. p is involved also in $SSS(\sigma_l)$. However, $M_{l-1}[p] \neq M_l[p]$ implies that $M(await(SSS(\sigma_l), l-1))[p] \neq M(await(SSS(\sigma_l), l))[p]$. Hence S is not independent. ■

It is worthwhile to note that this proof shows not only that an independent set is executable, but also that an independent set statically gives tight upper bounds on the number of tokens that can accumulate at each place in the entire PN. Specifically, the bound of a place p is given by the maximum number of tokens at p over all the markings associated with the nodes of the schedules in which p is involved. If the place corresponds to a communication channel, this bound determines the size of the channel.

This observation validates the following scheduling flow: 1) derive SS schedules for uncontrollable source transitions, 2) check them for independence, and 3) if the set is independent, the schedules can be executed with tight upper bounds on the accumulation of tokens at places.

Class of specifications having independent sets of schedules is practically significant. This is formally stated by the following proposition. It claims that for the class of PNs obtained in our case, we can always obtain an independent set of schedules, and thus the step 2 of the scheduling flow above is not necessary.

Proposition 4.3 *For a PN generated from FlowC as defined in Section 3, any set of single-source schedules is independent.*

Proof: Suppose the opposite. Then there exists a place p which is involved in two SS schedules, e.g. $SSS(a)$ and $SSS(b)$. Let this place p be the first place in $SSS(a)$ with such property. Namely, if v is the first node (in traversal of $SSS(a)$) such that tokens of p are consumed by the transition associated with an edge out of v, then for any other place p' involved in $SSS(b)$ and any node w traversed before v, transitions of the ECS of w do not consume tokens from p'. Let us consider a predecessor transition c of p in $SSS(a)$, i.e. $M(u) \xrightarrow{c} M(v)$.

Case 1. Transition c is involved in $SSS(b)$.

Then any predecessor place p' of c is also involved in $SSS(b)$. This does not satisfy the conditions of choice for p (node u precedes node v) and contradicts the assumption of Case 1.

Case 2. $SSS(b)$ involves a transition d which is a predecessor of p and $d \neq c$.

Let us consider the origin of place p in the considered PN.

a) Place p is an internal place for some process.

Then its predecessor transitions belong to the same process. All transitions of $SSS(a)$ preceding c are not involved in $SSS(b)$ by the choice of p. Therefore all of them are independent from transitions in $SSS(b)$. From this it follows that c and d are concurrent. The latter contradicts the conditions of sequentiality for each process, as given in Section 3.

b) Place p is a port place for some process.

From the point-to-point communications immediately follows that p must have predecessor transitions from the same process. Then following the arguments from item a) one can conclude that c and d being independent should be concurrent which contradicts the nature of the process. The latter disproves Case 2. ■

4.4 Termination conditions in exploring the schedules

A single source schedule is derived by exploring a subset of a reachability graph of a given PN. Existence of source transitions in the PN makes its reachability graph infinite. The latter leaves few hopes for its exhaustive exploration.

Pruning the search space is possible by identifying some search directions as non-promising. This could be done either based on formal criteria or heuristically. The largest class of PNs, for which the schedulability problem is known to be solvable exactly, are marked graphs that specify only non-choice behaviors. For them by solving the set of simultaneous linear equations one could find a t-invariant which gives a set of transitions necessary to fire to return to the initial marking. The schedule could be extracted from possibly infinite reachability graph through checking its finite subset defined by the set of transitions from t-invariants [2].

The step from non-choice to choice behaviors is known to be drastically more complex. For specifications modeled by Boolean Dataflow (BDF), the scheduling problem becomes undecidable [3]. Contrary to BDF, data-dependent choices in PNs are handled conservatively since data values to resolve choices are not taken into account in the scheduling, and hence the undecidability result for BDF does not necessarily imply the undecidability of that problem for general (non Unique-Choice) Petri Nets. Rather, either

- our approach is conservative (it can classify as un-schedulable some schedulable specification), or
- our problem is also undecidable (we are unaware of formal results on the decidability of PN scheduling, and the problem seems to be very difficult),

or both.

In the rest of this section we discuss conservative heuristic approaches for pruning the exploration of reachability space while constructing a schedule.

A straightforward way to limit the considered PN markings is given by requiring a pre-defined bound on each place. This defines a finite set of reachable markings in which the token counts are smaller than the place bounds. If a schedule within pre-defined place bounds exists it could be found through an exhaustive traversal of a subset of reachability graph implied by bounded markings [13]. If no schedule is found then either the PN has no schedule or the place bounds are too tight and a schedule might exist in a bigger subset of reachability space. The main advantage of this approach is in its simplicity because checking the constraints (bounds) can be done in an easy way during the reachability graph traversal. The difficult part, of course,

is how to manually impose the bounds a priori. Sometimes they can be extracted from the semantics of the specification, but there is no general method for doing that in arbitrary PNs. Moreover, as we discuss further, local (marking-based) criteria for bounds calculation cannot be applied, since schedulability may depend also on the history of a marking in the scheduling tree[4].

We have developed another approach for pruning the search space in the construction of a schedule which is based on the notion of *irrelevant markings*. The definition of irrelevant markings proceeds in two steps: 1) bounds on places are calculated from the structure of the PN, 2) while traversing the reachability graph, a marking is discarded (is considered irrelevant) if it covers some preceding marking in the graph and exceeds bounds on places. The second condition implies that our approach is significantly different from that of [13], that is based on pre-defined place bounds, because the decision on discarding marking from consideration is not local but requires checking the marking pre-history.

Definition 4.4 (Place degree) *The degree of a PN place p is the maximum of:*

a) max_weight(input(p)) + max_weight(output(p)) -1,

where max_weight(input(p)) *and* max_weight(output(p)) *are the maximal weights among input and output arcs of p (respectively), and*

b) *the number of tokens in p under the initial marking.*

In other words, $degree(p) = \max(\max_{t \in \bullet p} F(t,p) + \max_{t \in p\bullet} F(p,t) - 1\ , M_0(p))$, where $\bullet p$ and $p\bullet$ denote the sets of predecessors and successors of p respectively.

Place degree is connected intuitively to the idea of "saturating" p with tokens. If the token count of p is $max\_weight(output(p))$ or more, then adding tokens to p cannot help in enabling successor transitions of p. Therefore further accumulation of tokens in p is not reasonable unless it pumps tokens to other non-saturated places. Note that the largest non-saturated marking of p is $max\_weight(output(p)) - 1$. Adding tokens to p at that marking is still meaningful for enabling successor transitions of p. By the firing of a single predecessor transition of p, at most $max\_weight(input(p))$ tokens can be added, thus justifying the expression for place degree as $max\_weigh(input(p)) + max\_weight(output(p)) - 1$.

Definition 4.5 (Irrelevant marking) *A reachable marking M is said to be irrelevant with respect to a reachability tree rooted in the initial marking M_0 if the tree contains marking $M1$ such that:*

(a) M is reachable from $M1$,

(b) no place has more tokens in $M1$ than in M, and

(c) for every place p at which M has more tokens than $M1$, the number of tokens in $M1$ is equal to or greater than the degree of p.

The example in Figure 7 illustrates the crucial difference between the approaches targeted to pre-defined place bounds and irrelevant markings.

The maximal place degree in PN from Figure 7(a) is k. This information is the best one can extract from the PN structure about place bounds and it might be useful as the guidance in assigning upper bounds. The upper bounds should clearly exceed place degrees. In fact, the higher place degrees are, the higher upper

[4]This history-dependence is also the reason why standard reachability tree-based arguments to prove the decidability of the schedulability problem for general PNs cannot be applied, since they rely only on the local token count.

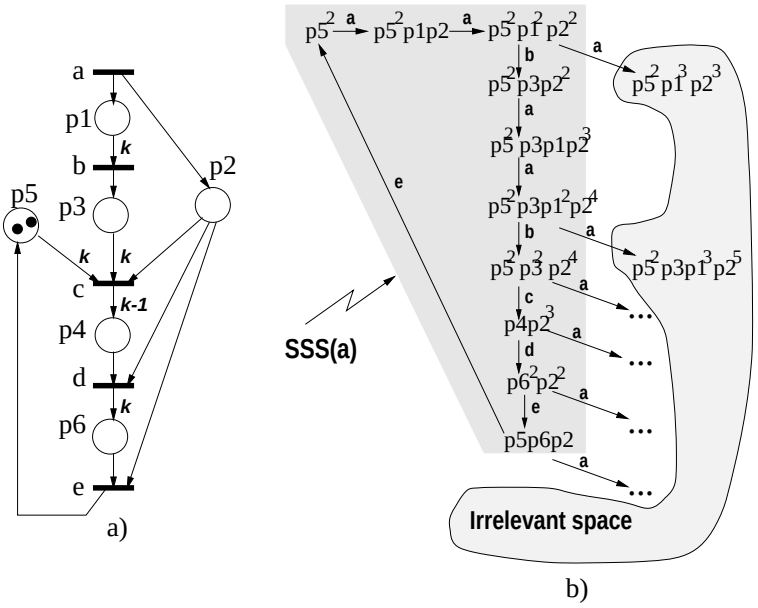


Figure 7: Constraining the search space for schedule based on irrelevance criterion

bounds are expected. Suppose that based on this rationale the upper bounds are chosen as maximal place degree multiplied by some constant margin.

Let us assume for our example that place bounds are assigned to be $2k-1$ and consider the PN reachability space when $k = 2$. When the schedule is checked with the pruning based on pre-defined place bounds any marking that has more than 3 tokens in a place should be discarded. Clearly no schedule could be found in that reachability space because after a, a, b, a occurs the only enabled transition is a but its firing produces 4 tokens in place $p2$ (see the part of reachability graph shown in Figure 7(b)). The search fails.

The irrelevance criterion handles this problem more graciously. It guides the search for the "proper" direction in the reachability space by avoiding the irrelevant markings. The first guidance is given when marking $p5^2p1^2p2^2$ is reached. In that marking one need to choose which transitions a or b to fire from the enabled set. The firing of a however produces the marking $p5^2p1^3p2^3$ which is irrelevant because it covers $p5^2p1^2p2^2$, where places $p1$ and $p2$ are already saturated. Therefore transition b should be chosen to fire. After this, a fires two times, resulting in the marking $p5^2p3p1^2p2^4$. Note that even though the place degree for $p2$ is exceeded in this marking, the marking is not irrelevant because in all the preceding markings containing $p3$, $p1$ 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 Figure 7(b)). Finally this procedure succeeds and finds a valid SS schedule.

A useful heuristics in traversing the reachability space based on the irrelevance criterion is to fire a source transition only when the system cannot fire anything else. This is reasonable because the smaller number of source transitions fire in the schedule, the smaller the schedule size is and the fewer await nodes it has. Using this heuristics to find a schedule for PN in Figure 7(a) allows us to decide unambiguously which transition to fire at any considered marking and results in $SSS(a)$ shown in Figure 7(b).

It is easy to see the superiority of our analysis, based on the irrelevance criterion, with respect to the analysis by place bounds of [13], because ours does not depend upon specific bound values, that can be shown at least in one example *not to exist* for an otherwise schedulable PN. Note that in Figure 7(a) transitions b

and c serve as consecutive dividers by k of the number of firings of transition a while d and e are consecutive multipliers by k-1 and k of the number of firings of transition c. Choosing longer sets of transitions to divide firings of a (with further multiplications) will violate *any* a priori constant bound. Contrary to that, the irrelevance criterion easily handles any chain of dividers to find a schedule.

Though pruning the search by using irrelevance seems a more justified criterion than by using place bounds, it is not exact for general PNs. There exist PNs for which any possible schedule enters the irrelevance space. This is due to the fact that for general PNs, the successors of a choice place may belong to more than one ECS, which may be enabled simultaneously at a given reachable marking. In this case, different subsets of those ECS's may become enabled by further accumulating tokens in the choice place beyond its degree. For UCPNs, adding tokens to a choice place does not change the nature of the choice, i.e. for an equal choice, either all the successors are enabled or none is enabled, while for a unique choice, always at most one of its successors is enabled. This gives the rationale of our *conjecture that the irrelevant criterion is exact for UCPNs*. However we are unable either to prove the exactness of this criterion or to find a counterexample for that. This issue is open for the moment.

5 Algorithm for Finding Schedules

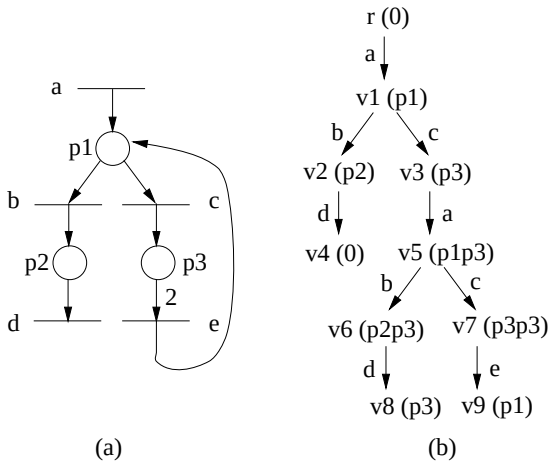
In this section, we describe an algorithm for finding a schedule of an uncontrollable source transition a of a given Petri net. The algorithm gradually creates a rooted tree. The root r satisfies the first and second properties of a schedule defined in Section 4.1. Each node and edge of the tree satisfies the third and fourth properties respectively. At the end of the algorithm, a cycle is created at each leaf, so that the resulting graph satisfies the fifth property.

Several notions used in the algorithm are first described in Section 5.1, and the algorithm is presented in Section 5.2. Various termination conditions can be adopted in the algorithm, each defining a search space in which schedules are sought. We illustrate how the algorithm works using examples in Section 5.3. Section 5.4 claims properties of the algorithm. We first show that if the algorithm terminates successfully, the graph returned by the algorithm is a schedule. We also show that if there is a schedule in a space defined by the termination conditions used, the algorithm always finds one. In Section 5.5, we first present an additional constraint to be incorporated in the proposed algorithm so that it generates only a single source schedule. The constraint can be viewed as a part of the more general problem: which ECS the algorithm first tries to associate with a given node of the schedule being created. Some heuristics on this problem are described also in Section 5.5, in order to gain efficiency of the algorithm in practice.

5.1 Preliminaries

Given two nodes u and v in a rooted directed tree, u is an *ancestor* of v, denoted by $u \leq v$, if a node u is on the path from the root to a node v. If in addition $u \neq v$, u is a *proper ancestor* of v, denoted by $u < v$. Further, for a given set V of nodes in which each pair is related with respect to the relation $\leq$, $\min V$ denotes a node u of V such that $u \leq v$ holds for each node v of V. u is called the *minimum* over V with respect to $\leq$.

An *entering point* u of a node v is an ancestor of v. Intuitively, the marking $M(v)$ associated with v satisfies the property that there is an ECS enabled at $M(v)$ such that for each transition t of the ECS, there is a sequence of transitions starting from t that can be fired from $M(v)$ and the marking obtained after the firing is $M(u)$. Further, this property holds at every marking obtained during the firing of the sequence. If an entering point of the root r is r itself, it implies an existence of a schedule in which regardless of transitions fired at the ECS's defined at the nodes, it is always possible to reach $M(r)$ from any node of the schedule.



(a) (b)

Node	r	v1	v2	v3	v4	v5	v6	v7	v8	v9
Entering point	r	r	r	v1	r	v1	v3	v1	v3	v1

(c)

Figure 8: Consider a rooted tree shown in (b), generated for the Petri net given in (a). The marking associated with each node is shown in parentheses adjacent to the name of the node. (c) presents entering points of the nodes. For the node v_9, v_1 is the unique entering point, because the associated markings are identical. Similarly, the unique entering points of v_8 and v_4 are given by v_3 and r, respectively. At v_7, an ECS $\{e\}$ is enabled. The child node of v_7 for this ECS is v_9 and the entering point of v_9 is v_1. Thus, the entering point of the ECS is v_1, which is also an entering point of v_7. For v_5, an ECS $\{b, c\}$ is enabled. The children of v_5 with respect to this ECS are v_6 and v_7, for which v_3 and v_1 are entering points respectively. Since $v_1 < v_3$, v_1 becomes an entering point of the ECS. Therefore, an entering point of v_5 is v_1. Indeed, starting from the marking $M(v_5)$, the marking $M(v_1)$ is reached by sequences ce and $bdace$, respectively. Thus, the marking of v_1 can be obtained by firing any transition of the ECS defined at v_5.

Formally, entering points are recursively defined on a node and an ECS. For a leaf v of a rooted tree, if there is a node u such that $u < v$ and $M(u) = M(v)$, the minimum over all such u is the unique entering point of v. If no such u exists, but there is an ECS enabled at $M(v)$ whose entering point is defined, then any entering point of such an ECS is an entering point of v. Otherwise, an entering point of v is undefined. For a leaf v and an ECS enabled at $M(v)$, consider a rooted tree obtained by creating a node w and a directed edge $[v, w]$ out of v for each transition t of the ECS. We set $M(w)$ so that $M(v) \xrightarrow{t} M(w)$ holds. An entering point is defined for the ECS, only if there exists a set V with the following property: V contains exactly one entering point for each child w of v, say $EP(w)$, and $EP(w) \leq v$ holds. In this case, the minimum over any such a set V is an entering point of this ECS. See Figure 8 for an example and further explanation.

5.2 The Algorithm

The algorithm takes as input a Petri net and a uncontrollable source transition a for which a schedule is sought. The core of the algorithm consists of two functions, `EP`(v, $target$) and `EP_ECS`(E, v, $target$), which are called each other. We explain these two functions in detail in the succeeding paragraphs. The overall algorithm works as follows. It first initializes a tree by creating the root r and set $M(r)$ to the initial marking of the Petri net. It further creates a node v and an edge $[r, v]$. We associate the transition a with this edge, and a marking with v so that $M(r) \xrightarrow{a} M(v)$ holds. The algorithm then calls the function `EP`(v, r). If this function returns the root r, a schedule has been found and the algorithm calls a post-processing and terminates successfully. The post-processing will be described at the end of this sub-section. Otherwise, the algorithm reports no schedule for a and terminates.

Figure 9 presents pseudo code of the two core functions. `EP` takes as input two nodes v and $target$ of the current tree, where we maintain the invariance that v is a leaf and $target < v$. The objective of this function

```
function EP(v, target)
  if(termination conditions hold) return UNDEF;
  if(∃u : u < v and M(u) = M(v)) return u;
  EP ← UNDEF, ECS(v) ← φ;
  for(each ECS E enabled at M(v))
    EP_ECS ← EP_ECS(E, v, target);
    if(EP_ECS ≤ target)
      ECS(v) ← E, return EP_ECS;
    if(EP = UNDEF or EP_ECS < EP)
      ECS(v) ← E, EP ← EP_ECS;
  return EP;
```

(a)

```
function EP_ECS(E, v, target)
  EP_ECS ← UNDEF, current_target ← target;
  for(each transition t of E)
    create a node w and an edge [v, w];
    T([v, w]) ← t;
    M(w) ← the marking obtained by firing t at M(v);
    EP ← EP(w, current_target);
    if(EP = UNDEF or v < EP) return UNDEF;
    EP_ECS ← min(EP_ECS, EP);
    if(EP_ECS ≤ target) current_target ← v;
  return EP_ECS;
```

(b)

Figure 9: (a) Pseudo code of EP(v, $target$). v is a leaf of the current tree, while $target$ is a proper ancestor of v. (b) Pseudo code of EP_ECS(E, v, $target$). E is an ECS enabled at the marking associated with v. $T([v, w])$ denotes the transition of the Petri net associated with the edge $[v, w]$. $current\_target$ is initially set to $target$, and is changed to v as soon as one entering point is found for some w such that the point is an ancestor of $target$. This is because our objective is to find an entering point of E which is an ancestor of $target$, and according to the definition, once we find one such a point for some w, we only need from other nodes entering points that are ancestors of v.

is to find an entering point of v that is an ancestor of $target$, if exists. It first checks termination conditions and if they hold, the algorithm ceases to search a schedule beyond this node, and the function returns a special value UNDEF. Examples of a termination condition are the irrelevance criterion introduced in Section 4.4, i.e. the condition holds if v is an irrelevant node ($M(v)$ is an irrelevant marking), or bounds on the places of the Petri nets. In the latter case, a positive integer called *bound* may be associated with a place p, and the termination condition holds at v if there exists a place for which $M(v)[p]$ is greater than the bound of p.

If the termination conditions do not hold, the function checks whether there is a proper ancestor u of v such that $M(u) = M(v)$. If this is the case, EP returns u. Otherwise, it searches an entering point for each ECS E enabled at $M(v)$ by calling EP_ECS(E, v, $target$). If it finds one that is an ancestor of $target$, it returns one such a point. If no such a point is found, it returns the minimum among all the entering points found. In either case, $ECS(v)$ is set to the ECS whose entering point is returned. If no entering point is found for any ECS enabled, UNDEF is returned and $ECS(v)$ is set to empty.

In EP_ECS(E, v, $target$), for each transition t of the ECS E, a node w and an edge $[v, w]$ are created. A marking is associated with w so that $M(v) \xrightarrow{t} M(w)$ holds, and t is associated with $[v, w]$. The objective of the function is to find an entering point of E that is an ancestor of $target$. For each w, the function EP is called to find an entering point of w. If EP returns UNDEF or a node which is not an ancestor of v, EP_ECS immediately returns UNDEF, since an entering point of E is undefined in this case. Otherwise, it returns the minimum over the entering points found.

The post-processing consists of two parts. First, we retain only a part of the created tree that are used in the resulting schedule, and delete the rest. The root and its child are retained, and a node w is retained if (a) its parent v is retained and (b) the transition $T([v, w])$ belongs to the ECS $ECS(v)$ that the function EP associated with v [5]. The second part of the post-processing creates a cycle for each leaf w of the retained portion of the tree, by merging w with its proper ancestor u such that $M(u) = M(w)$. By construction,

[5] In our actual implementation, this process is done dynamically in EP.

such a u exists, since otherwise EP would have returned UNDEF for w, as w is a leaf, but this contradicts the fact that for the parent v of w, $T([v,w])$ belongs to the ECS $ECS(v)$ associated with v. Further, such u is unique, since EP does not continue the procedure if such u is found. Finally, the graph obtained at the end is returned.

5.3 Example

We illustrate the algorithm for the Petri net shown in Figure 8-(a), where the irrelevance criterion is used as the termination condition. Initially, the algorithm creates nodes r and v_1, and an edge $[r, v_1]$ with the associated transition a. $M(v_1)$ is set to p_1, and EP(v_1, r) is called.

Since v_1 is not an irrelevant node, the termination condition does not hold. ECS's enabled at $M(v_1)$ are $\{b, c\}$ and $\{a\}$. Suppose that $\{b, c\}$ is first chosen, and EP_ECS($\{b, c\}$, v_1, r) is called. In EP_ECS, suppose that b is processed first, and a node v_2 and an edge $[v_1, v_2]$ are created. We set $M(v_2) = p_2$ and $T([v_1, v_2]) = b$, and call EP(v_2, r).

EP(v_2, r) computes ECS's enabled at $M(v_2)$: $\{a\}$ and $\{d\}$. Suppose that $\{a\}$ is chosen first and EP_ECS($\{a\}$, v_2, r) is called. EP_ECS creates a node v_{10} and an edge $[v_2, v_{10}]$, and sets $M(v_{10}) = p_1p_2$. The graph obtained at this moment is shown in Figure 10-(a). It then calls EP(v_{10}, r).

ECS's enabled at $M(v_{10})$ are $\{d\}$, $\{b, c\}$, and $\{a\}$. Suppose that EP(v_{10}, r) processes the ECS's in this order. When EP_ECS($\{d\}$, v_{10}, r) is called, a node v_{11} is created, $M(v_{11})$ is set to p_1, and EP(v_{11}, r) is called. Since v_1 is a proper ancestor of v_{11} and $M(v_1) = M(v_{11})$, EP(v_{11}, r) does not continue further and returns v_1. With this, EP_ECS($\{d\}$, v_{10}, r) also finishes by returning v_1.

We have now come back to EP(v_{10}, r). For the value v_1 just obtained for the ECS $\{d\}$, $v_1 \leq target$ does not hold. Thus, we update EP to v_1, and try the next ECS: $\{b, c\}$. In EP_ECS($\{b, c\}$, v_{10}, r), suppose that b is chosen first and a node v_{12} is created with $M(v_{12}) = p_2p_2$. When EP(v_{12}, r) is called, we find that v_{12} is an irrelevant node, since v_2 is its proper ancestor and the marking at v_2 satisfies the degree of p_2. Thus, EP returns UNDEF to EP_ECS($\{b, c\}$, v_{10}, r), which in turn returns UNDEF to EP(v_{10}, r).

We have again come back to EP(v_{10}, r). Since the value just obtained is UNDEF, we move on to the last ECS: $\{a\}$. As with the case of $\{b, c\}$, the algorithm will find that the node created for this ECS is an irrelevant node, and we will receive UNDEF again. Since all the enabled ECS's have been processed, EP(v_{10}, r) returns the current value of EP, which is v_1. The graph obtained at this point is shown in Figure 10-(b).

The algorithm is now back at EP(v_2, r). The value just received is v_1. Since $v_1 \leq target$ does not hold, we set $EP = v_1$ and continue for the ECS $\{d\}$. In EP_ECS($\{d\}$, v_2, r), a node v_4 and an edge $[v_2, v_4]$ are created. We set $M(v_4) = 0$, and call EP(v_4, r). Since $M(v_4)$ is the initial marking, associated with the root r, EP immediately returns r. Thus, EP_ECS($\{d\}$, v_2, r) also returns r. EP(v_2, r) receives this value, and since $r \leq target$ holds, it returns r while setting $ECS(v_2) = \{d\}$.

The algorithm is now at EP_ECS($\{b, c\}$, v_1, r). It sets EP_ECS to the just received value r. Since $r \leq target$ holds, $current\_value$ is set to v_1. We then process the other transition c of the ECS, by creating a node v_3 and an edge $[v_1, v_3]$. Then EP(v_3, v_1) is called. Note that $target$ is changed from r to v_1. If the algorithm proceeds to create a graph as shown by the subtree rooted at v_3 in Figure 8-(b), EP(v_3, v_1) returns v_1. The graph obtained at this moment is shown in Figure 10-(c). With this, EP_ECS($\{b, c\}$, v_1, r) completes its procedure, and returns r. Hence, the algorithm comes back to EP(v_1, r), which returns r and the post-processing is called.

The first part of the post-processing deletes the nodes v_{10}, v_{11}, v_{12}, and v_{13}, as well as their in-coming edges. The second part then creates a cycle for each leaf of the resulting tree, and the algorithm finally generates a graph shown in Figure 10-(d).

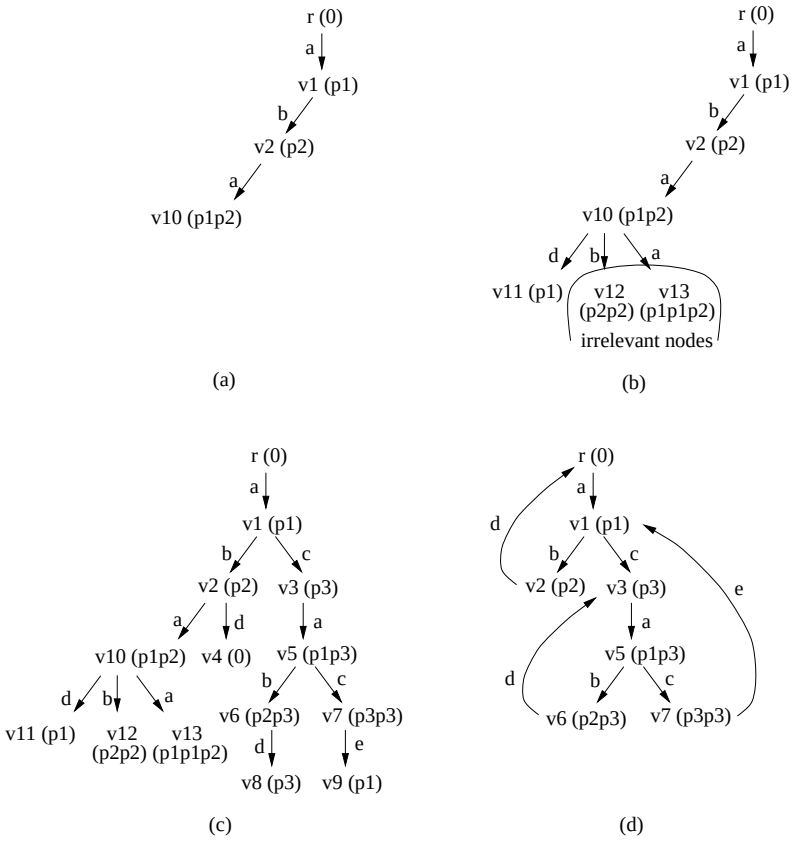


Figure 10: Illustration of the Proposed Algorithm for the Petri Net given in Figure 8-(a).

5.4 Effectiveness

We first show that if the algorithm terminates successfully, it returns a schedule.

Theorem 5.1 *Suppose that the algorithm terminates successfully, when applied for a source transition a. Then the returned graph is a schedule for a.*

Proof: We show that the returned graph satisfies the five properties of a schedule given in its definition in Section 4.1. All but the fifth hold by construction, where the node r created as the root in the beginning of the algorithm is the distinguished node. We focus on the fifth property. As the node r has a directed path to each node of the graph, we show that each node has a directed path to r. To prove this, it is sufficient to claim that if we denote by u the node returned by the function `EP` when applied to a node v, the graph has a directed path from the parent of v to u. This is because this property holds transitively and u was a proper ancestor of v in the constructed tree.

We prove the claim by induction. First, consider a node v that was a leaf in the tree obtained just before the post-processing was called. Let u be the node returned by `EP` for v. Since v was a leaf, it follows that $u < v$ and $M(u) = M(v)$. Since the post-processing created an edge from the parent of v to u, the claim holds for v. Now, for a given node v, let us employ the induction hypothesis that the claim holds for every node w such that an edge $[v, w]$ existed in the tree obtained just before the post-processing was called. Let u be the node returned by `EP` when called for v. By construction of `EP`, u was returned by the function `EP_ECS` when called with the ECS given by $ECS(v)$. It follows that there was a node w in the tree obtained just before the post-processing such that an edge $[v, w]$ existed in the tree and u was returned by `EP` when applied to w. By the induction hypothesis, a directed path exists from v to u in the resulting graph, and thus the claim holds. ■

The next question we consider is whether the algorithm can find a schedule for a transition a, if there exists one. The answer depends on termination conditions used in the function `EP`, since they define the search space of the algorithm. We characterize the search space using the reachability tree of the Petri net. To do this, let us first introduce several assumptions and definitions.

We assume that a termination condition can be defined as a boolean function τ which takes as input a node of the reachability tree and returns 1 if and only if the condition holds. The irrelevance criterion and bounds on places, the two conditions given as an example of termination conditions in Section 5.2, can be defined in this way. Without loss of generality, we assume that a termination condition does not hold at the root.

For a given termination condition τ, let RT_τ be the maximum connected subtree of the reachability tree with the following three properties: (1) RT_τ contains the root, (2) $\tau(v) = 0$ for each node v in RT_τ, and (3) if $u < v$ and $M(u) = M(v)$ for two nodes u and v of RT_τ, v is a leaf. Intuitively, RT_τ is obtained by traversing the reachability tree from the root, where the traversal is terminated at a node v if $\tau(v) = 1$ or there exists $u < v$ such that $M(u) = M(v)$. RT_τ is defined by the nodes visited in this traversal except those at which τ holds.

We say that a schedule for a is *contained* in RT_τ, if RT_τ contains a subtree at the root that can simulate the schedule, i.e. no matter how one traverses the schedule, it is possible to traverse the subtree, retaining the same sequence of transitions and markings. More formally, a function f can be defined from the nodes of the subtree to the nodes of the schedule with three properties. First, f maps the root of RT_τ to the distinguished node of the schedule. Second, for each node v of the schedule, there is a node v' such that $v = f(v')$ and $M(v) = M(v')$. Third, for each edge $[v, w]$ of the schedule, each non-leaf node v' of RT_τ with $v = f(v')$ has an edge $[v', w']$ such that $w = f(w')$ and the same transition is associated with the both edges. We call f a containment function for the schedule.

The following shows that RT_τ is the search space of the algorithm for a termination condition τ.

Theorem 5.2 *For a given termination condition τ, suppose that RT_τ is finite. The algorithm terminates successfully when applied for a source transition a, if and only if there exists a schedule for a contained in RT_τ.*

Proof: We first note that the rooted tree created by EP and EP_ECS is isomorphic to a subtree of the reachability tree of the Petri net. The isomorphic function φ is defined so that the root is mapped to the root of the reachability tree, and each edge $[v, w]$ is mapped to $[\varphi(v), \varphi(w)]$ with the same transition associated. Thus, we say that EP is applied to a node v of the reachability tree, by which we mean that EP_ECS creates a node that is mapped by φ to v and EP is applied to this node.

If the algorithm terminates successfully, Theorem 5.1 shows that the resulting graph is a schedule. This schedule is contained in RT_τ, where the subtree of RT_τ that simulates the schedule is defined by the range of the isomorphic function φ. A containment function for the schedule is given by the inverse of φ.

Let S be a schedule for a contained in RT_τ. Let f be the containment function for S such that no subfunction of f, induced by a strict subset of the domain of f, satisfies the all three conditions of the definition of a containment function. We claim that for each node v of the subtree of RT_τ defined by the domain of f, if EP is applied to v during the algorithm, its returned value is a proper ancestor of v. More specifically, we show that (1) there exist u and l in the subtree such that $u < v$, $v \leq l$, and $M(u) = M(l)$, and (2) for all such u and l, the returned value of EP(v, $target$) is an ancestor of either u or $target$. The subtree contains the child of the root r of RT_τ, for which EP is always applied by construction of the algorithm. Since r is the only proper ancestor of the node, EP returns r and the algorithm terminates successfully.

First, if v is a leaf of the subtree, it is a leaf of RT_τ that has a proper ancestor u with $M(u) = M(v)$, and the claim holds. Let v be a non-leaf node of the subtree, and suppose that the claim holds for every child of v in the subtree. Suppose that EP(v, $target$) is called. By the fifth property of the definition of a schedule, $f(v)$ is on a directed cycle that includes $f(r)$. Thus there exist u and l such that $u < v$, $v < l$, and $M(u) = M(l)$. Since v and r are in the subtree, so are u and l. Consider arbitrary u and l with the property. For each child w of v in the subtree, $u < w$ and $w \leq l$. By the induction hypothesis, if EP(w, $target$) is applied to w, its returned value is an ancestor of either u or $target$. Since the set of transitions associated with each of these edges $[v, w]$ is an ECS, say E, it follows that if EP_ECS(E, v, $target$) is called, its returned value EP_ECS is an ancestor of either u or $target$. If EP does not call EP_ECS for E, then there exists another ECS enabled at $M(v)$ to which EP_ECS is called, and the returned value is an ancestor of $target$. Hence, the claim holds. ■

5.5 Sorting ECS's

5.5.1 Generating Single Source Schedules

A single source schedule for an uncontrollable source transition a does not involve any uncontrollable source transition other than a. Finding such schedules is important, since a Petri net obtained from a FlowC specification has a property that every set of single source schedules becomes independent, and the independence guarantees the executability of the schedules, as shown in Section 4.3.

In order to guarantee that the proposed algorithm generates only a single source schedule, we do not call EP_ECS for the ECS's of other uncontrollable source transitions in the function EP. As a result, the algorithm reports no schedule if there is no single source schedule for a, even though there may exist a schedule for a which involves other uncontrollable source transitions.

5.5.2 Heuristics using T-invariants

To find an entry point of a node, the algorithm may explore all possible ECS's at the node until it finds a desired one. The number of nodes created in the algorithm depends on the order of ECS's explored. Although the ordering does not influence the worst-case search space or run time of the algorithm, some orderings help finding a desired entry point sooner than others. In this section, we present a heuristic approach of sorting the ECS's for this purpose. The heuristic helps keeping the resulting schedules small. It also provides us with a sufficient non-schedulability condition, and if the condition holds, we can immediately terminate the procedure, reporting no schedule.

The heuristic method tries to find a short sequence of transitions such that if the sequence is fired from the current node, a marking associated with some ancestor of the node can be obtained. Such an ancestor becomes a candidate of an entry point returned by the function `EP` for the node.

We use t-invariants in finding such a sequence. A t-invariant is a vector of non-negative integers that solves the system of homogeneous marking equations $Cx = 0$, where C is the incidence matrix of the Petri net. 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 i-th place and j-th transition respectively. A t-invariant represents a set of sequences in which the number of occurrences of the j-th transition is given by the integer at the j-th position of the t-invariant. We say that such a sequence is *contained* in the t-invariant. Each of such sequences has a property that if the sequence can be fired from a marking M, the marking obtained after the firing is also M. We call a non-negative basis of the homogeneous marking equations a base of t-invariants. Such a basis can be obtained by using a Smith Normal Form decomposition [11].

It is known that a schedule does not exist if there is no base of t-invariants. Therefore, if the algorithm identifies this case, it terminates immediately without applying the function `EP`.

If a base of t-invariants is found, using the base and a sequence of transitions associated with the path from the root to the current node, the heuristic computes a vector of non-negative integers called the *promising vector* in `EP`. The positions of the vector correspond to the transitions, and those with positive integers represent transitions to be fired from the current node. The ECS's are sorted using this vector. Specifically, the modified `EP` works as follows.

`EP` takes three additional inputs, besides the two inputs originally described in Section 5.2. The first is a base of t-invariants, represented by a matrix in which each row specifies a t-invariant. The second is a matrix that we call *partial invariant matrix*. Initially, it is identical with the base of t-invariants, and is going to be modified as the algorithm proceeds. The third is the promising vector, which is initially set to a null vector (a vector of all 0's).

Once the end of the third line has been reached in the pseudo code of `EP` in Figure 9-(a), we compute ECS's enabled at the marking associated with the current node. If there is none, `UNDEF` is returned. If there are enabled ECS's, we check if at least one of them has a transition that appears in the promising vector, where we say that a transition *appears* in a vector if the vector has a positive integer at the position for the transition. If not, we find a new promising vector, as explained in the succeeding paragraphs. At this point we have a promising vector and a set of enabled ECS's. We sort the enabled ECS's so that for a given pair of ECS's, if one of them has at least one transition that appears in the vector and the other ECS has no such a transition, the former is positioned before the latter. We then call `EP_ECS` with the ECS according to the established order.

`EP_ECS` also takes as input the promising vector and the two matrices provided as additional inputs to `EP`. In `EP_ECS`, when a new node is created for a transition t, the column position corresponding to t is decremented by one in the partial invariant matrix. If the column has -1 at some row after this operation, we transform each of such row vectors into a non-negative one by adding another row vector of the base

of t-invariants. This means that the resulting matrix represents t-invariants given by linear combinations of the base, from which the transitions fired so far from the root are subtracted. We keep track of the subset of t-invariants of the base that have been added to each row vector. The matrix is then passed to `EP`. The promising vector is decremented in the same way and is passed to `EP` as well, where if the position for t becomes -1, we pass a null vector.

We now explain how to compute a promising vector in `EP`. Our objective is to find a vector that is fireable at the current marking $M(v)$, so that the marking associated with some ancestor can be reached after the firing. We say that a vector of transitions is fireable at a marking M if the vector contains a sequence of transitions that can be fired at M. To compute such a vector, we use a t-invariant given by a linear combination of the base of t-invariants. However, a known problem with t-invariants is that it is in general difficult to identify whether a t-invariant is fireable at a given marking [11]. Nevertheless, due to the structure of a Petri net generated from a FlowC specification, a necessary condition can be obtained for a vector to be fireable. To describe the condition, let us introduce some terminology. Consider a place p that does not correspond to a channel in the original network specified in FlowC. If p is marked at a marking M, an ECS that a successor of p belongs to is called a *pseudo-enabled* ECS at M. By the process of such an ECS, we mean the process of the network to which this ECS belongs. Since a pseudo-enabled ECS does not contain a source transition, such a process is uniquely defined. 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 5.3 *For any vector of transitions fireable at a reachable marking M in a Petri net obtained from a FlowC specification, and for any pseudo-enabled ECS at M, if the process of the ECS appears in the vector, the ECS has a transition that appears in the vector.*

Proof: Suppose for contrary that there exists a pseudo-enabled ECS at M, say E, such that its process appears in the vector but no transition of E 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 E. Since a Petri net created for the process by the FlowC compiler has the property that there is exactly one place p marked at M, the ECS containing t_1 is not pseudo-enabled at M. Further, if a sequence of transitions is fireable at M and the marking reached after the firing enables t_1, the sequence must contain a transition of E. Since no transition of E appears in the vector, each sequence of transitions in the vector contains t_1 but none of the transitions of E. Thus the sequence is not fireable at M. ■

Our procedure first finds a subset of the base of t-invariants such that the sum of the t-invariants in the subset satisfies this necessary condition at $M(v)$. The subset, which we call the candidate invariant, is used to compute a promising vector. If no such a subset exists, we use an empty subset. The problem can be formulated as the binate covering problem. Consider a matrix A such that columns correspond to the invariants of the base. A row of A is defined for each pair made of a pseudo-enabled ECS at $M(v)$ and an invariant b of the base such that the process of the ECS appears in b but none of the transitions of the ECS appears in b. The row has 0 at a given column if the invariant corresponding to the column is b, 1 if the ECS contains a transition that appears in the invariant corresponding to the column, 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 $A_{ij} = 1$. It follows that the subset of the base given by a feasible solution satisfies the necessary condition Theorem 5.3. We employ a method given in [10] that always finds a feasible solution, if exists, while the cardinality of the subset is heuristically made minimum.

In computing a promising vector, we choose a row vector in the partial invariant matrix that is closest to the candidate invariant. First we check if there is one such that the subset of the t-invariants in the base corresponding to the row vector contains the candidate invariant. If this is the case, we choose as the promising vector the one with the minimum cardinality of the subset. Otherwise, we choose a row vector with the minimum number of t-invariants of the base that are not common with the candidate invariant. The promising vector is given by adding to this row vector the t-invariants in the candidate invariant that have not been added to the row vector.

When the enabled ECS's are sorted using the promising vector as described, one may employ additional heuristics to break ties. We present three such heuristics.

The first is on the termination condition employed in the algorithm. For a given pair of ECS's, suppose that one ECS has a transition such that when `EP_ECS` creates a child w of v for the transition, the termination condition holds for w. If the other ECS does not have such a transition, the latter is favored over the former.

The second is on uncontrollable source transitions. If one ECS consists of an uncontrollable source, while the other does not, the latter is favored. The rationale is that the system has to wait for the environment to fire an uncontrollable source transition, and we would like to avoid such a transition in a schedule if possible.

The third is to disfavor an ECS with more than one transitions. The rationale is that since such an ECS creates more than one edges out of the current node, it is likely that ECS's with single transitions appear in all the paths out of the node. It is empirically found that using ECS's with single transitions first helps finding a smaller schedule.

6 Code Generation

The goal of code generation is to synthesize a sequential program to be run on a microprocessor which implements the system behavior, based on the schedule computed with the algorithm presented in Section 5. Definition 4.3, Proposition 4.3 and Theorem 5.2 guarantee that we are always able to generate independent Single Source Schedules for a *FlowC* specification, if it exists in the reachability space RT_τ defined by a termination condition τ (e.g., irrelevance or pre-defined place bounds): with this hypothesis, the problem of generating n tasks can be decomposed into n independent problems of synthesizing a single task, one for each uncontrollable source transition, given its schedule. Therefore, in the following, we focus on generating code for a single uncontrollable source transition, rather than for the whole system.

The Petri Net theory has been used so far to model processes and schedules. In this section we will describe how those models can be translated into a software implementation. Places and transitions of a Petri Net, and nodes and arcs of a schedule all have a correspondence in the generated code. In particular, the number of tokens in a place is represented through a state variable, a transition is translated into the corresponding code, a node of the schedule is the state of the system, corresponding to a particular configuration of the state variables (marking), and the sequence of ECSs in a path in the schedule gives the sequence of transitions in the synthesized code; if an ECS has more than one transition, a run-time data dependent choice shall be generated.

Although a direct translation of the schedule into code is possible, this would be very inefficient with respect to the memory required to store it, since many transitions are duplicated. Hence we perform optimizations in successive steps in order to minimize it. The generated code has an added structure to reflect the schedule, written in the C language. Of course, the various generated tasks still communicate among them and with the environment via read and write primitives, and thus they assume the availability of a run-time scheduler and an implementation of those primitives, e.g., as buffers (see in Section 8).

First, some terminology for the code generation algorithm is introduced in Section 6.1. Then the algorithm for traversing a schedule is presented in Section 6.2. Handling of I/O ports is examined in Section 6.3. Post-processing optimizations and synthesis of the final code are described in Section 6.4. The same example used in Section 5.3 is shown in Sections 6.2.1 and 6.4.4.

6.1 Terminology

Some notions, which were not introduced before in this paper, are needed to properly describe the code generation algorithm and are described here. For other terms, we refer the reader to Sections 2 and 4.

Thread Let $S(a)$ be a schedule for an uncontrollable source transition a. Let N be the set of nodes in $S(a)$ and $W \subseteq N$ be the set of await nodes in $S(a)$. A *thread* is a subgraph $TH(v,a) \subseteq S(a)$ with the following properties:

1. $v \in W$ is an await node in $S(a)$ such that a is the ECS associated with the outgoing edges of v.
2. $v \in TH(v,a)$
3. Let $w \in N$; if $\exists u \in N$ such that $[u,w] \in S(a)$, $u \notin W \setminus \{v\}$, $u \in TH(v,a) \Longrightarrow w \in TH(v,a)$.

We denote with $M(TH(v,a)) = M(v)$ the initial marking of a thread. Intuitively, a thread $TH(v,a)$ is a subgraph of a schedule which starts from an await node v and includes all nodes reachable from it, until it hits a different await node.

Thread equivalence Given two nodes n_1 and n_2 of a schedule S, they are said to be equivalent[6], and we write $n_1 \sim n_2$, if and only if $ECS(n_1) = ECS(n_2)$. Given two threads $TH_1(v_1,a)$ and $TH_2(v_2,a)$, they are said to be equivalent, and we write $TH_1(v_1,a) \sim TH_2(v_2,a)$ if and only if there exist a one to one function f from the nodes of TH_1 on to the nodes of TH_2 such that:

1. $f(v_1) = v_2$,
2. $\forall u_i \in TH_1(v_1,a) : u_i \sim f(u_i)$
3. $\forall [u_i,u_j] \in TH_1(v_1,a) : [f(u_i), f(u_j)] \in TH_2(v_2,a)$

Intuitively, two threads are equivalent if the corresponding graphs are identical, with the same transitions on the edges, but possibly with different markings in the nodes. Execution of two equivalent threads leads to firing the same transitions, but the final await node can be different.

Often two threads are not equivalent, but a subgraph of them satisfies the previous properties. In this case we say that they are partially equivalent, and the subgraph is a candidate to become a code segment (explained later in this section).

[6] Our definition of thread equivalence is close to classical Labeled Transition System bi-simulation equivalence.

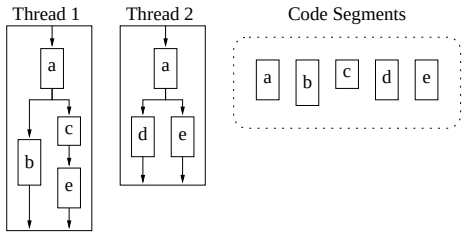


Figure 11: Correspondence between threads and code segments

Task Let S be a set of schedules, a an uncontrollable source transition and $TH_k(v_k, i)$ a generic thread for uncontrollable source i . Then we say that $T(a)$ is a task for a if and only if:

$$T(a) = \bigcup_{TH_k} TH_k(v_k, i) \text{ such that } i = a$$

Therefore a task contains all the threads related to a particular uncontrollable source transition, covering all the possible behaviors defined by the set of schedules.

Code segment A *code segment* is a directed rooted tree that associates a transition with each edge so that for each node v', the set of transitions at the edges out of v' is an ECS, denoted by *ECS*(v'). Each node of a code segment maintains a set of pairs made of marking and an ECS: each pair is called a *state* of the node. We establish a correspondence between nodes of a code segment and nodes of schedules for which we generate code. More than one node of a schedule may correspond to a node of a code segment. The states of a node of a code segment represent the markings and ECSs of the corresponding nodes of the schedules.

Code segments are used to represent portions of threads, and therefore are subgraphs of the schedule, where the information associated with each node is augmented for the purpose of synthesis. The rationale is to save code size: often many threads in a task are equivalent, either completely, or in a smaller portion; if code is synthesized for each thread, then a lot of unnecessary replications will be generated. Therefore, it is useful to determine their common shared portions: a *code segment* is the largest sequence of transitions and/or choices common to one or more threads in the same task. Given a set of n non equivalent threads, there are in general at least n code segments, and more if some part of the code is shared.

Therefore, as depicted in Figure 11, a thread may be composed of more than one code segment, and each code segment may appear in one or more threads. In the example shown, code segments `a` and `e` are shared between the two threads, while `b`, `c` and `d` are not.

The sequence of transitions to fire within a code segment does not depend on which thread is actually executing, since it depends only on data dependent choices. On the contrary, the sequence of code segments to call to react to an occurrence of a source transition depends on which thread should be selected, which in turn depends on the await node reached on the previous reaction.

6.2 Algorithm for schedule traversal

The first step in code generation is traversing the schedule to find all the threads that make up a task. The product of this step is a collection of graphs (code segments) and a data structure which represent the general organization of the final implementation. Threads are not explicitly stored, but the sequence of code segments that realize them can be easily determined by looking at the state (marking and ECS) of each code segment

leaf.

The traversal generates a minimal set of code segments with the property that for each node v of $S(a)$, the set has exactly one node v' with $ECS(v) = ECS(v')$. We say that such a v corresponds to v'. Since both v and v' have the same ECS, v' can be used during synthesis to generate an implementation of that ECS; moreover, since there is only one node with a given ECS in the set of code segments, the code size is also minimized. Given a node v' of a code segment, the state includes $(M(v), ECS(v))$, for all v in $S(a)$ which correspond to v'. If v' is a leaf, then for each node u of $S(a)$ that corresponds to the parent u' of v', $(M(v), ECS(v))$ is included in the state set, where v is the child of u such that the transition $T([u, v])$ is associated with $[u', v']$. The last property is equivalent to stating that the schedule $S(a)$ is made acyclic by replicating the destination nodes of the loops.

Intuitively, the ECS at each node of a code segment is used to determine the sequence of transitions, and hence of statements, to execute. ECSs with more than one transition also need a run-time data dependent choice to select the correct one. Intuitively, the state information represents the decisions taken by the scheduler to always find a way to schedule transitions, no matter what the environment does. States are kept for every node in a code segment during schedule traversal. However, since the execution flow of transitions does not depend on the state within a code segment[7], only the initial and final states are considered for code synthesis.

There are two main functions, `traverse` and `compare`. The first one prepares the data structure and start the traversal, by calling `compare`, from the root node of the schedule. The second one recursively compares the schedule to existing code segments, or creates a new code segment if needed. As an initialization step, the single source schedule is analyzed to cut the loops; this is necessary to find a termination condition for the traversal. However, in order for the traversal to completely visit the graph, the property that each node has a path from the distinguished node r must be preserved. This is achieved by construction, starting a depth first traversal of the schedule from r, and cutting loops when already visited nodes are encountered: a leaf is then created with the same marking and enabled ECS of the destination node of the loop. The fifth property of a schedule defined in Section 4.1 guarantees that a path from r exists in the original graph, and it is preserved by this processing.

The pseudo-code for function `traverse` is shown in Figure 12. We assume here that an uncontrollable source transition has already been selected and that the argument s is the corresponding single source schedule. Two lists are used in this function:

`nodelist` : a set of nodes of the schedule to be used as the starting point of a recursive comparison.

`codelist` : a list of the code segments which make up the threads and the task of the source transition.

At the beginning the `codelist` is obviously empty and it should be filled by the traversal. The `nodelist` initially contains only the root node of the schedule, which will be used as the first starting point; the function `compare` will then add new nodes to `nodelist` during its operation. `traverse` gets and removes the first element of `nodelist` and checks if it is equivalent to the root of any already existing code segment: if it is not, a new entry in `codelist` is created with a root node equivalent to the node from the schedule, and a flag (`newcodeseg`) is set. The function `compare` is then called, either to fill the newly created code segment, or to compare an existing one to a subgraph of the schedule rooted at the considered node.

Figure 13 is the pseudo-code for function `compare`. Its goal is to examine a single thread, compare it to existing code segments and create them if needed. At the end, the final await nodes reached are put in

[7] It may well depend on data dependent choices, but these will be resolved at run-time and do not represent a state.

```
function traverse(s) {
    nodelist = Create();
    codelist = Create();
    appendElem(nodelist, root(s));
    while (!empty(nodelist)) {
        node_sch = getRemoveFirstElem(nodelist);
        newcodeseg = FALSE;
        if (!(node_code = findCodeSeg(node_sch, codelist))) {
            newcodeseg = TRUE;
            node_code = initCodeSeg(node_sch, codelist);
        }
        compare(node_sch, node_code, newcodeseg);
    }
}
```

Figure 12: Pseudo-code for function `traverse`

`nodelist` so that new invocations of the function can proceed with the traversal of the entire schedule. In some cases, however, the function returns before reaching an await node, if certain conditions, which will be detailed in the following, occur.

This function works by calling itself recursively, and implements a depth first search of the thread in the schedule graph. The argument `node_sch` is a node in the schedule, and `node_code` is the node in a code segment to be compared. Note that even if a code segment is new, at least one always exists because it is created by the `traverse` function before calling `compare`.

There are two distinctive sections, depending on if a code segment already exists or if it shall be created. If it is new (the flag `newcodeseg` is set), then new children are created in the code segment by copying the children from the node of the schedule and properly setting their state (marking and ECS). This can be done because it is guaranteed that the two parent nodes in the code segment and in the schedule are equivalent and therefore have the same ECS. Then, for each child in the schedule, termination conditions for the traversal are checked: it will stop if the child has no other children (it means we have reached a loop in the schedule), or if it has children but the associated ECS is a source transition or a code segment already exists with the root node with the same ECS. In the last two cases, the child is also added to `nodelist` for further traversal. Sometimes, an ECS is found which is already associated with a node of a code segment, but it is not at the root: then a new code segment is created, by cutting the existing one at that node, an making it the new root (function `Detach`); moreover, if the considered child had children, it is also added to `nodelist`. If none of the previous conditions occur, the traversal will continue recursively.

If the code segment we are comparing to is not new, then the marking and the ECS of its node are updated by adding those of the node in the schedule to the state set. If the ECS is identical to all the previous ones, it means that the node in the thread and that in the code segment are equivalent, and therefore the comparison can continue. However, it should be stopped if the node in the code segment does not have children, and the one in the thread does, as we should select another code segment for comparison. On the contrary, if the ECS is different from the previous ones stored in the node in the code segment, a leave has been identified; if there are children in the code segment, then they should be detached to create a new code segment, and the node in the schedule is added to `nodelist`. Note that no check is needed for source transitions in the schedule to identify the end of a thread, since it is guaranteed that when a code segment is created, it is cut at await

```
function compare(node_sch, node_code, newcodeseg) {
    if (newcodeseg) {
        createKids(node_code, Kids(node_sch));
        Foreach_kid(node_sch, &kid) {
            if (nKids(kid) != 0)
                if (isSource(ECS(kid)) || findCodeSeg(kid, codelist))
                    appendElem(nodelist, kid);
                else if (oldnode_code = isInCodeSeg(ECS(kid), codelist)) {
                    Detach(Kids(oldnode_code));
                    appendElem(nodelist, kid);
                } else
                    compare(kid, Kid(node_code), newcodeseg);
            else    /* (nKids(kid) == 0) */
                if (oldnode_code = isInCodeSeg(ECS(kid), codelist))
                    Detach(Kids(oldnode_code));
        }
    } else {   /* NOT newcodeseg */
        updateState(node_sch, node_code);
        updateECS(node_sch, node_code);
        if (ECSequal(node_code)) {
            if (nKids(node_code) == 0 && nKids(node_sch) != 0)
                appendElem(nodelist, node_sch);
            else
                Foreach_kid(node_sch, &kid)
                    compare(kid, Kid(node_code), newcodeseg);
        } else {   /* NOT ECSequal */
            if (nKids(node_code) != 0) Detach(Kids(node_code));
            appendElem(nodelist, node_sch);
        }
    }
}
```

Figure 13: Pseudo-code for function `compare`

nodes.

Given a schedule $S(a)$ for an uncontrollable source transition and the set of code segments $CS(a) = \{CS_1(a), \ldots, CS_k(a)\}$ derived with this algorithm for the task, some properties hold by construction:

- For each node of the schedule there is one and only one node in the code segments which is equivalent to it. The converse is not true.

  $\forall n \in S(a)\ \exists u \in CS(a) : u \sim n$ and $\forall w \in CS(a) : w \neq u \Rightarrow w \not\sim n$.

- If two nodes in the schedule are equivalent, then they correspond to the same node in the code segments.

  $\forall n_1, n_2 \in S(a)\ n_1 \sim u, u \in CS(a) \Rightarrow n_2 \sim u$.

The first property shows that the entire schedule is translated into code segments and no part of it is lost during processing. The second property guarantees that code size is minimized, since equivalent nodes are never duplicated.

6.2.1 Example of schedule traversal

We show how the algorithm performs on the schedule presented in Figure 10-(d), with the loops on transitions d and e cut, as in Figure 8-(b). The results are presented in Figure 14, where for each node the marking and the ECS are indicated in parenthesis, separated by a comma. We assume that nodes in the code segments are denoted by u_i, and nodes in the schedule by v_i.

At the first call we must create a new code segment (cs_1, see Figure 14-(a)), which in this case corresponds to an entire thread. The function `compare` stops at node u_4 because the corresponding node v_4 in the schedule has no more children[8], and at node u_3 because the next transition in the schedule is a source transition (*ECS*$(v_3) = a$). Node v_3 is put in the `nodelist` at the end.

The second traversal starting at v_3 does not need to create a new code segment, since one beginning with transition a already exists (function `findCodeSeg` in `traverse` returns u_0). During the comparison new markings and ECS are added for each node of the code segment; the traversal stops again at u_4 (v_8 in the schedule) because, even if both ECS are equal to a, node v_8 has no children and the procedure will exit without calling `compare` recursively. It also stops at u_3 (v_7 in the schedule) because the two ECS are different (one is a, the other is e); there is no need to call the function `Detach` since u_3 has no children, and v_7 is added to `nodelist`. Thus, the second thread is not complete, and the algorithm continues. The result of this step is shown in Figure 14-(b).

The third (and last) traversal creates a new code segment cs_2, starting with transition e, and it finishes immediately because node v_9 has no children. Since v_9 was created by cutting a loop, its ECS is equal to that of the destination node of the loop, which is v_1 in this case (see Figure 10-(d)). Therefore, *ECS*$(v_9) = \{b, c\}$, which already exists in cs_1, node u_1; hence function `Detach` is called which creates code segment cs_3 rooted at u_1, and cuts cs_1. No more nodes are added to `nodelist`, so the traversal is finished. The final three code segments are shown in Figure 14-(c).

From the state information contained in the nodes of each cs_i, it is possible to draw a directed graph for the threads, having code segments as nodes. Every thread starts with cs_1, which always contains as the first transition the uncontrollable source; the number of threads is determined by how many marking and ECS pairs can be found in the state set of root node of cs_1, and the graph is obtained by following each of them separately, knowing that the states are always ordered in the same way within a code segment. Therefore, in

[8] Note that function `isInCodeSeg` returns *null* for node v_4 because root node of code segments are excluded from the search.

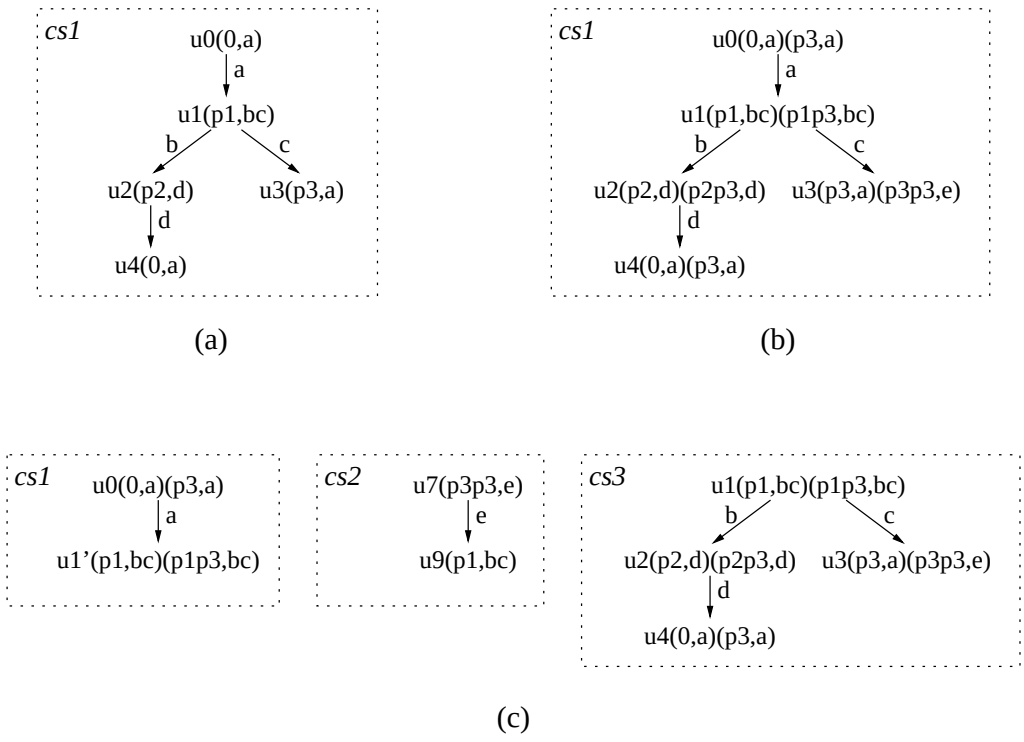


Figure 14: Example of schedule traversal and creation of code segments

the example presented there are two threads TH_1 and TH_2, and the two graphs are shown in Figure 15. By inspecting the initial and final state information of each code segment, it is easy to see that it is sufficient to only know the number of tokens in place p_3 to completely determine the behavior to implement as a reaction to the occurrence of source transition a.

6.3 I/O ports handling

Channels can disappear because of processes collapsing into one task. These channels, called *intra-task* channels, should be handled in a special way, by using local arrays for communication. A circular buffer can be used, and there is no need to check for overflow because the scheduler already takes care of it. If the array has size 1, then it can also be substituted by a normal variable of the proper type.

Other channels, like those that connect to the environment or between tasks, do not need any special handling and the corresponding code can be synthesized as in the original specification. Consequently, communication primitives are written in FlowC in the generated code. For implementation, when an operating system is chosen, it may be necessary to represent the primitives using API provided by the operating system itself, as explained more in Section 8.1.

6.4 Code synthesis

The goal of synthesis is to take the output of the traversal and generate code in a target language from it. Since the association between transitions and code is already known, a structure should be added to reflect the schedule, and variables should be used to keep track of the state. The syntax of both the structure and

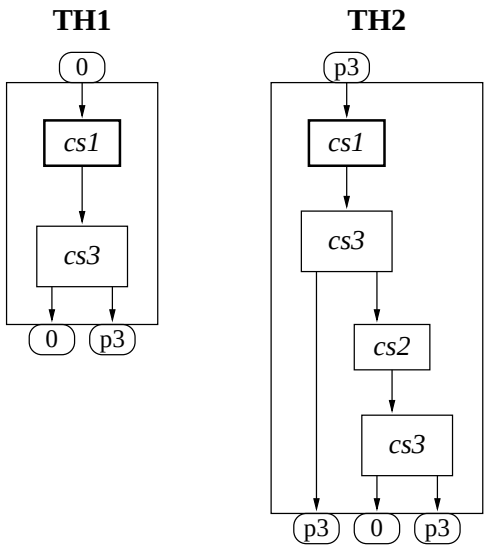


Figure 15: Graphs for threads using code segments

the variables is language dependent, and given the similarities between FlowC and C, we have chosen the latter to implement them. The rest of the code will still be in FlowC, so another step is required if one wants to compile it on a microprocessor, as described in Section 8. The decision to support only C is not seen as a limitation, because C is the main language used by designer for implementation, and others can be added with very little effort.

With the hypothesis of independent single source schedules it is guaranteed that there is no interaction between tasks, and so we can generate one independent specification for each of them. In the following we will therefore concentrate on a single task synthesis, which is divided into three main parts: declaration (Section 6.4.1), initialization (Section 6.4.2) and run (Section 6.4.3).

6.4.1 Declarations

This part includes several declarations needed in the C language, such as new data types, prototypes and global variables. These may be divided into two broad classes:

1. Declarations from the original (FlowC) specification. They are stored by the compilation and linking processes into a file, which can be simply included. Note that local variables of a process become global variables, whose name is made unique during linking.

2. Declarations needed for the schedule structure. These include in particular the state variables, which correspond to places in the Petri Net. Note that not all places are needed, since only a subset of them is sufficient to completely determine the current state. This subset is computed before starting the actual synthesis by looking at which places are updated by code segments, and which places are needed for conditions. The intersection of these two sets defines the set of state variables.

This part is also used to redefine macros for `WRITE_DATA` and `READ_DATA` statements whose channel disappeared due to the collapse of processes into a single task (intra-task channels).

Note that when multiple tasks are generated, each of them will have separate state variables whose value is not shared. However, places in the Petri Net can appear in any task and potentially the same one can

be part of the state of two different tasks. This can be a problem, since each of them will update the state locally, and that change will not reflect to the other one, leading to inconsistencies. Fortunately, in our case the hypothesis of single source independent schedule guarantees that the intersection between states of tasks is always empty, and therefore this kind of implementation is correct. If the hypothesis is not satisfied, then a more complicated scheme should be used.

6.4.2 Initializations

State variables need to be properly initialized so that the first reaction is correct. The value is derived from the scheduler, and for places of the Petri Net it is the initial marking (number of tokens), which is a positive integer. Being global variables, the value will not be lost between two successive executions, so it just need to be updated correctly by the task.

Also buffers for intra-task channels should be initialized; since at the beginning no information is available for such buffers, their value is set to zero. Also the pointers used for handling the buffers are initialized.

6.4.3 Run

This part generates the code to be used to implement a task, and includes all the threads by means of the code segments found during traversal. When a source transition occurs this function should be invoked, so its goal is to select the correct code segments to execute a thread, based on the current state (marking), and to update the state for the next fire.

The starting point of all the threads in a task is cs_1, which is the only one to contain the uncontrollable source transition, always associated to the edge between the first and the second node of the code segment. So cs_1 is the first to appear in the `Run` function; all the other code segments can appear in any order, since `goto`s statements are used to jump from one to another.

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

execution It contains the real code for transitions and data dependent choices, taken from the FlowC specification. It always starts with a label, which is the concatenation of the name of the transitions that form the ECS of the first node of the code segment. The label is used to jump to the code segment, and it is generated also for cs_1, although it is never used.

Then the graph for the code segment is traversed in a depth-first search manner. For each node whose ECS is a single transition, the code for that transition is copied from the output of the compiler and pasted into the output file. If the node is a choice, then an `if-then-else` construct is generated, where the condition is also taken from the compiler output. The TRUE and FALSE branches are identified by looking at the original Petri Net, which stores this information in arcs.

When a leaf is reached, the *update* and *jump* sections are generated before going back in the traversal, so that if there are choices in the code segment, more than one `goto` appear in the implementation.

No optimization is performed on the code for a transition. We are in fact more concerned about optimizing the schedule and communication rather than lower level issues that can be more efficiently handled by a compiler.

update At each leaf of a code segment the state must be properly updated so that:

1. the next code segment to call to complete the thread can be correctly selected,

2. the state at the end of the thread corresponds to the await node in the schedule reached by the execution.

If more than one code segment must be called for a given thread, each leaf will update only a portion of the state, that which was actually changed by the execution of the respective code segment. The sum of all of them constitutes the global state change, which corresponds to the difference in marking between the initial await node and the final one for the thread.

The execution of a code segment may change the number of tokens in a place of the Petri Net; however, only those which have a net increment or decrement different than zero are updated, and only if they concur to the definition of the state for that task. Otherwise, they are not interesting and are completely discarded to reduce memory size.

jump This section must find which code segment to call next, or should return if the thread is finished. With the exception of leaves, for all the nodes in a code segment the ECS associated with a set of state is always the same. For a leaf, this property is not true, and the ECS represents what to do next. Therefore, a switch construct on the state (marking) is used to select a `goto` statement, which will cause the execution to jump to the label named after the destination ECS. If the destination is cs_1, then a `return` is generated instead of a `goto`, because it means we have reached an await node and the thread has finished.

Some optimizations are performed to reduce the number of comparisons: for instance, if all the ECS are the same (see nodes u'_1, u_9 and u_4 in the example of Figure 14-(c)), the switch is not needed and a single `goto` is synthesized. If more than one ECS are equal, but not all, a condition is looked for to include them in just one jump. Otherwise, a full switch must be generated.

Synthesis will therefore generate a function called `ISR` (for Interrupt Service Routine, since the uncontrollable source transition should be handled in interrupt), which has no local variables and starts with code segment cs_1, followed by all the others in the order in which they were found during traversal. When the last code segment is generated, the function is closed. This function has just one entry point, but may have several exit points corresponding to all the leaves that perform a `return`.

Scheduling of transitions inside a task is static except when a data dependent choice is involved, which is always resolved at run-time. On the contrary, scheduling of the task should be dynamic: the ISR may or may not be called immediately when the source transition occurs: the operating system can execute it as soon as the interrupt is detected (privileging low response latencies) or decide to fire it at a later time (letting other processes to complete).

6.4.4 Example

We show what the code looks like for the example presented in Section 6.2.1. Since it was not derived from a FlowC source, not all the feature of code generation will be presented; for a more complete example see Section 8.

The first part is declaration of state variables, data types, ports and other variables. By looking at the code segments in Figure 14, it is easy to determine that the only place needed to specify the state is p_3, which should be an integer. Other declarations are to be found in a header file produced during compilation and linking of the FlowC sources, so we just include it. The final code for the declaration part is therefore shown in Figure 16, lines 1–2.

```
1   #include “example.data.h”
2   int p3;

3   p3 = 0;

4   void ISR(void)
5   {
6   a:
7       a();
8       goto bc;
9   e:
10      e();
11      p3 = p3 - 2;
12      goto bc;
13  bc:
14      if (condition(p1) == TRUE) {
15          b();
16          d();
17          return;
18      } else if (condition(p1) == FALSE) {
19          c();
20          p3 = p3 + 1;
21          if (p3 == 1) {
22              return;
23          } else if (p3 == 2) {
24              goto e;
25          }
26      }
27  }
```

Figure 16: Generated code for the example

Next, we need to initialize state variables and arrays for intra-task channels. In this case we don’t have any information about channels because we started directly from a Petri Net, without the FlowC source. Please refer to Section 8 where a discussion about this is presented. The only variable which needs initialization is therefore the state variable p_3: the initial marking of the Petri Net is 0, so the corresponding code is in line 3 of Figure 16.

Finally the function `ISR` must be synthesized. There are three code segments, the first one being cs_1, each identified with a label named after the ECS of the first node. For each transition, the code is indicated by the name of the transition itself, followed by a pair of brackets, like for a function call; when a FlowC source is available, this call is substituted with the actual code. Also, condition in places are taken from the compiler output, and the `TRUE` and `False` branches are annotated in the Petri Net. The function `ISR` is shown in Figure 16, from line 4 to line 27.

The first code segment cs_1 does not have the *update* section because no tokens are either read or written to place p_3. Segment cs_2 consumes two tokens instead, so the *update* section is present. Both segments have a *jump* section which is just a `goto`, since no check is needed to decide what to do next. Code segment cs_3

is more complicated because of a data dependent choice, whose code is contained in place p_1: depending on the outcome, one of the two branches is taken. The TRUE branch executes the code for transitions b and d, and then exits; the FALSE one executes c, updates the place p_3, and then jumps or returns depending on the value of the updated state.

The generated code cannot be compiled and run yet: first, FlowC primitives which do not have a direct implementation in C should be redefined in order to use some communication API. Second, there is no `main` function, so a scheduler, usually embedded in the Real Time Operating System (RTOS), should invoke the `ISR` to execute the code when the uncontrollable source transition occurs.

7 Other issues

7.1 Synchronization-dependent choice

The FlowC language as described in Section 3 does not allow one to specify control depending on the availability of tokens on input ports. This inability, on the one hand, has the advantage that the results of the computation (i.e. the stream of token values on primary output ports) does not depend on the schedule [6]. On the other hand, it either limits the kind of allowed specifications, or forces one to model the "absence" of tokens with a special token value, thus resulting in less efficient implementations. Our choice was to add to FlowC the `SELECT` construct [4], that can be used to identify which port has enough tokens to perform a `read` or `write` operation without blocking (the latter may block only when a channel has a user-defined bound). For example, assuming that `p0` and `p1` are input ports connected to channels that currently have 2 tokens each, and that `p2` is an output port connected to a channel without a user-given bound, the construct:

```
switch (SELECT (p0, 1, p1, 3, p2, 1024)) {
case 0: ... READ_DATA (p0, buffer, 1); ... break;
case 1: ... READ_DATA (p1, buffer, 3); ... break;
case 2: ... WRITE_DATA (p2, buffer, 1024); ...
```

non-deterministically selects *exactly one* of the statements labeled with either `0` (`p0` needs at least one token on its channel) or `2` (`p2` needs at least 1024 free positions on its channel). The statement labeled with `1` cannot be executed because `p1` needs at least 3 tokens. Due to its semantics, `SELECT` in this case ensures that none of the following `READ_DATA` or `WRITE_DATA` can block[9]. If no port can perform the required operation, then `SELECT` blocks until at least one becomes ready. In order to provide a more deterministic execution semantics, `SELECT` ports can also be given (at the netlist level) a priority that resolves between multiple enabled choices.

Addition of `SELECT` to FlowC has three very significant consequences, and should thus be used only when needed:

1) The resulting process network no longer has a behavior that is schedule-independent, thus verification after scheduling and architectural mapping become more difficult (functional, i.e. input-output equivalence of various schedulings can no longer be assumed [6]).

2) Now the designer can control which branch will be taken by a given process by using write statements in another block. This can be used, as discussed in the next section, to eliminate some data-dependent false paths, at the expense of some more complex control.

3) The underlying Petri net is no longer Unique Choice.

[9] This is true, of course, only if these statements require at most the same number of tokens or positions as those specified in `SELECT`, as in the example above.

7.2 False paths

Petri nets conspicuously ignore data values when modeling data-dependent choice. This is both an advantage, as it allows us to model data-dependent choice without immediately falling into the undecidability of Boolean Dataflow ([3]), and a disadvantage, since some perfectly schedulable specifications could be classified as unschedulable by our conservative heuristics.

Consider, for example, a pair of processes `A` and `B`, that communicate via two channels `c0` and `c1`, connected with ports of the same name of the two processes.

```
PROCESS A (...) {
  for (i = 0; i < 10; i++) WRITE_DATA (c0, buf1[i], 1);
  for (i = 0; i < 2; i++) READ_DATA (c1, buf2[i], 1);
}
PROCESS B (...) {
  for (i = 0; i < 10; i++) READ_DATA (c0, buf3[i], 1);
  for (i = 0; i < 2; i++) WRITE_DATA (c1, buf4[i], 1);
}
```

It should be clear that the two processes are quasi-statically schedulable, and their behavior is that of assigning 10 items of data from `buf1` to `buf3` and 2 items from `buf4` to `buf2`. However, when the FlowC source is translated to a PN, the data values are lost, and the resulting PN is not schedulable. For example, there is an unbounded growth of channel `c0` if `A` keeps writing while `B` stops reading. But this path (as all other deadlocking or overflowing paths in this example) is *false*, i.e., it can never be executed by the two processes due to data dependencies (in this case, the same fixed loop bounds; note that there are also potentially unbounded loops that exhibit similar behaviors).

Schedulability can be obtained via a judicious use of `SELECT` with priorities, e.g., by using this construct to terminate the "dependent" reading loops (without changing the behavior, or adding more synchronization constraints). We need to add a new pair of ports, `doneA` and `doneB`, with priorities *lower* than `c0` and `c1`, and change the processes as follows.

```
PROCESS A (...) {
  for (i = 0; i < 10; i++) WRITE_DATA (c0, buf1[i], 1);
  WRITE_DATA (done0, 0, 1); // tell B the loop is done
  // loop until there is data on done1 and nothing on c1
  for (i = done = 0; !done; i++) {
    switch (SELECT (c1, 1, done1, 1)) {
    // c1 has higher priority
      case 0: READ_DATA (c1, buf2[i], 1); break;
      case 1: READ_DATA (done1, d, 1); done = 1;
    }
  }
}
PROCESS B (...) {
  // loop until there is data on done0 and nothing on c0
  for (i = done = 0; !done; i++) {
    switch (SELECT (c0, 1, done0, 1)) {
    // c0 has higher priority
```

```
      case 0: READ_DATA (c0, buf3[i], 1); break;
      case 1: READ_DATA (done0, d, 1); done = 1;
    }
  }
  for (i = 0; i < 2; i++) WRITE_DATA (p2, buf4[i], 1);
  WRITE_DATA (done1, 0, 1); // tell A the loop is done
}
```

In this case, the scheduling algorithm is able to identify that the overflowing path is indeed *false*, i.e., it can never be executed by the two processes considered simultaneously. It produces a schedule that looks as follows:

```
{
  for (i = 0; i < 10; i++) buf3[i] = buf1[i];
  for (i = 0; i < 2; i++) buf2[i] = buf4[i];
}
```

We are currently exploring at ways of automatically transforming the first (un-schedulable) form of FlowC into the second (in this case schedulable) one, based on "hints" from the designer, who can label loops and tell the FlowC compiler that their executions in different processes should be "synchronized" using `SELECT`[10].

Unfortunately, though, undecidability of quasi-static scheduling for Boolean Dataflow networks implies that even by using `SELECT` there will be perfectly schedulable specifications that are not accepted by our *conservative* algorithm. We hope that it can schedule most *interesting* specifications, even though of course we will never be able to formally prove it.

8 Implementation and Experiments

We have implemented the entire flow from *FlowC* sources to synthesized tasks in a set of tools, which comprise compiler, linker, scheduler and code generator. These tools have been used to apply the methodology presented in this paper to an example taken from a multimedia application.

8.1 Code import in VCC

We targeted a C-based language, called *Poindexter C*, supported by Cadence VCC codesign environment ([9]) for function specifications. This enable us to simulate, both functionally and with performance estimation, the synthesized system. The generated code, as described in Section 6, still needs some changes in order to be compiled and executed, as VCC communication primitives should be used in place of `READ_DATA` and `WRITE_DATA`, and correct structure should be given to the Poindexter C source.

Poindexter C is a language used to describe the behavior of a software block, and therefore has primitives to define input and output ports and perform communication on them. Two distinctive functions specify what happens when the block is initialized (`poin_entry_Init`) and when it is fired (`poin_entry_Run`). The

[10] The technique exemplified above does not work if the sender contains two loops writing both to the same channel, since that channel would not be empty by the time `done` is written, and hence `SELECT` would never choose `done`. However, a simple modification based on writing to different channels from adjacent loops can be used.

```
void poin_entry_Run(void) {
    if (controllable_port1.Enabled()) {
        addToken(array1, controllable_port1.Value());
    }
    ...
    if (controllable_portn.Enabled()) {
        addToken(arrayn, controllable_portn.Value());
    }
    if (uncontrollable_port.Enabled()) {
        addToken(array, uncontrollable_port.Value());
        ISR();
    }
}
```

Figure 17: Code for function `poin_entry_Run`

main issues in translating the generated code into Poindexter C thus refer to correctly use the primitives and fill the functions.

Each task is translated into a single Poindexter block: all input and output channels become ports for the block, except for intra-task channels, which are completely hidden from the interface and the corresponding read and write primitives operate on local arrays (or variable if they are of unit size). Standard VCC API, like `port_Enabled()`, `port_Value()` and `port_Post()`, are used to read and write ports. However, direct translation of FlowC communication primitives to those API is not always possible, as explained in the following.

VCC model of computation assumes that a block is fired whenever an input port receives a token, if resources are available. This is not the behavior we want, since a reaction is needed only if an uncontrollable port is involved; the solution is to call the `ISR` from the `poin_entry_Run` function only when the uncontrollable port has tokens. Since we want to be able to read data from controllable ports whenever the schedule requires it, we also need to latch those values when they arrive; therefore, the controllable ports are checked for enabledness first, and any data found is stored in a temporary array. Also the uncontrollable port value is latched before calling the `ISR`, so all the `READ_DATA` become a read operation on arrays. Hence, the structure of the `poin_entry_Run` function, as shown in Figure 17, checks all the uncontrollable ports for tokens, latches the values, and calls the `ISR` only when needed.

Write operations also needs special handling. Opposite to the read case, buffering is needed only for intra-task channels; however, since Poindexter C does not support sending multiple tokens in a single invocation of a `Post`, writing to an output port cannot be directly translated unless only one piece of data is involved. It therefore requires a loop, and for each iteration a token is written to the output port. The number of iteration is determined dynamically by the value specified in the last parameter of `WRITE_DATA`.

Once it is synthesized, the Poindexter C specification is read into VCC using its *import* facility. It is stored into a user defined library and the symbol is automatically generated. It can then be used in any behavioral schematic for both functional and performance simulation.

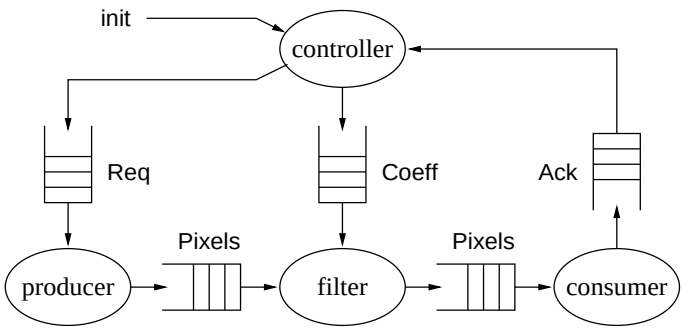


Figure 18: Process network for a video application

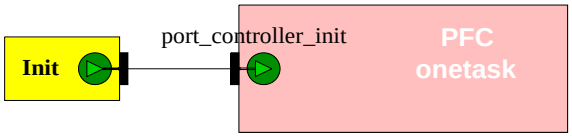


Figure 19: VCC behavioral diagram for the PFC example

8.2 Experiment

The system we used for the experiments is made of four processes, and is schematically depicted in Figure 18. It implements a video application where a `producer` generates image data, `filter` processes them given some coefficients and `consumer` reads the final image. The process `controller` governs the whole system, and is triggered by `init`, the only uncontrolled port. All the process are potentially concurrent and communicate through FIFO channels. The producer, filter, consumer chain constitutes the *hard real-time* video data path, while the controller makes up the *soft real-time* control path. The system shows multiple data rates, as the pixels can be transmitted either one by one, grouped into a line, or in an entire frame. Also, the coefficients for filtering are read (using SELECT) only if available, otherwise the ones received for the previous frame are used.

We have described the whole system in FlowC, using four processes. The specification does not have all the details about the image generation, filtering and display, but very simple algorithms have been used instead. After compilation and linking, our proposed algorithm generated, in less than a minute, a single task with all the channels of unit size.

The synthesized code has been imported into VCC and functionally simulated to verify the correctness of the algorithm. The behavioral diagram is shown in Figure 19. Print statements have been manually added to compare the execution with the four process system, and the output was exactly the same.

Performance comparisons between the single synthesized task and the original four process system, where each process is implemented as a separate tasks executed by a simple round-robin scheduler, has been carried out on a real R3000 machine, through compilation and profiling. Rather than using function calls to implement communication in the four processes case, we decided to inline them: we obtained a significant increase in performance (on average 30%) but also an increase in code size.

We performed several experiments varying the size of the channels, the number of frames transmitted. and the optimization options of the compiler used to generate the final implementation. The results of comparison between our single-task implementation and the one in which each original process is implemented as a separate task are shown in Figure 20. The y axis reports the number of clock cycles and the x axis

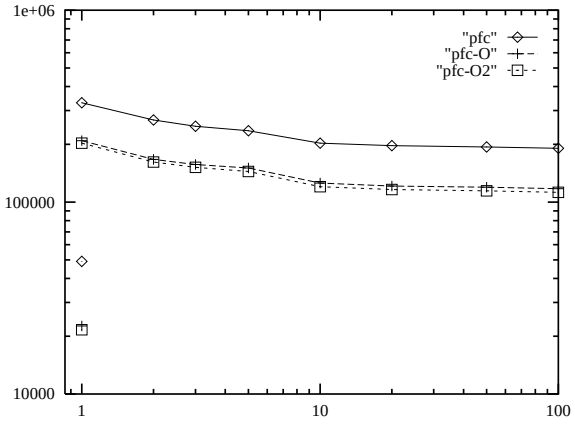


Figure 20: Execution time and memory requirements for single task versus four tasks (with various buffer sizes).

	pfc			pfc-O			pfc-O2		
	1 task	4 procs	*ratio*	1 task	4 procs	*ratio*	1 task	4 procs	*ratio*
10	49	190	3.9	23	117	5.1	22	112	5.1
50	245	950	3.9	113	586	5.2	108	560	5.2
100	490	1900	3.9	226	1171	5.2	216	1120	5.2
500	2450	9492	3.9	1131	5851	5.2	1080	5596	5.2
1000	4900	18984	3.9	2261	11703	5.2	2160	11193	5.2

Table 1: Experimental results with different number of frames

the size of the channel buffers; the number of frames transmitted was set to 10. The three lines represent the four-task version under different compiler options: large buffers clearly improve performance, but also increase the memory needed to implement the system. Frames were made by 10 lines of 10 pixels each, so a buffer size equal or greater than 10 gives a little boost in performance since an entire line fits in it. The three dots in the lower left corner represent the performance of the single generated task, which always uses one place buffers as determined by our scheduler. The result of our procedure out-performs by a factor of 4 to 10.

For the experiments with different numbers of frames, our single-task implementation was consistently faster by a factor of 4 to 5, where frame counts were changed between 10 and 1000. In all cases, the four process system was implemented using buffers of size 100 to obtain a faster execution. The results are summarized in Table 1, where all performance data are in thousands of clock cycles. (except the columns labelled *ratio*).

Finally, we compared the code size for the two systems, with different compiler optimizations. We excluded from the total the size of the RTOS and static data, which is very high especially in the 4 process system when the buffers are large. With inlined communication primitives, the single generated task is around 7 times smaller; if function calls are used, it is still 3 times smaller, but performance of the 4 process system are very much degraded.

The experimental results on code size are presented in Table 2, which shows that the size of the single task is significantly smaller than the total size of the 4-task version, with inline communication primitives. However, note that this comparison is a little biased in favor of the single task implementation: our approach is extremely good in reducing code size when a lot of communication is involved, but cannot do much with complex algorithms embedded inside a transition, which were not present in out example. Moreover, we

	1 task	4 procs					ratio
		contr	*prod*	*filt*	*cons*	*total*	
pfc	912	1020	1468	2463	1616	6567	7.2
pfc-O	416	560	836	1360	860	3616	8.7
pfc-O2	400	556	804	1300	812	3472	8.7

Table 2: Experimental results for code size. All values (except *ratio*) are in bytes.

looked at object size only, hence the overhead due to the linking phase in the compilation process, which is the same for both implementation, is not included.

9 Conclusions

We proposed a procedure for synthesizing a set of software tasks from a concurrent functional specification. The specification is given as a network of concurrent processes, where each process performs sequential computation that may involve data-dependent controls. A task is generated for each port that receives a trigger from the environment, and its sequential program is obtained as a schedule of operations to be executed for the port. Our procedure synthesizes schedules that can be executed in finite memory. Only the data-dependent controls need to be resolved at run-time. Experimental results show that tasks generated by our approach can reduce run-time overhead significantly, compared to the case where each process of the specification is implemented as a separate task.

References

[1] S. Amellal and B. Kaminska. Functional synthesis of digital systems with TASS. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 13(5), May 1994.

[2] S. Bhattacharyya, P. Murthy, and E. Lee. *Software synthesis from dataflow graphs*. Kluwer Academic Press, 1996.

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

[4] 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. Submitted to the Design Automation Conference 2000, June 2000.

[5] H. Gomma. *Software design methods for concurrent and real-time systems*. Addison-Wesley Publishing Company, 1996.

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

[7] G. Lakshminarayana, K. Khouri, and N. Jha. Wavehead: A novel scheduling technique for control-flow intensive designs. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 18(5), May 1999.

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

[9] G. Martin and B. Salefski. Methodology and technology for design of communications and multimedia products via system-level IP integration. In *ACM/IEEE Design, Automation and Test in Europe (DATE)*, February 1998.

[10] H. Mathony. Universal Logic Design Algorithm and its Application to the Synthesis of Two-level Switching Circuits. *IEE Proceedings*, 136 Pt. E(3), May 1989.

[11] T. Murata. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE*, 74(4), April 1989.

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

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

[14] F. Thoen, M. Cornero, G. Goossens, and H De Man. Real-time multi-tasking in software synthesis for information processing systems. In *Proceedings of the International System Synthesis Symposium*, 1995.