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


Jordi Cortadella
Universitat Politècnica de Catalunya

Alex Kondratyev
Theseus Logic

Luciano Lavagno
Università di Udine

Marc Massot
Universitat de Girona

Sandra Moral
Universitat Politècnica de Catalunya

Claudio Passerone
Politecnico di Torino

Yosinori Watanabe
Cadence Berkeley Labs

Alberto Sangiovanni-Vincentelli
University of California Berkeley



## Abstract

The problem of optimal software synthesis for concurrent processes to be implemented on a single processor is addressed. The approach calls for the representation of the concurrent processes with Petri nets that give a theoretical foundation for the scheduling algorithm that sequentializes the concurrent processes and for the code generation step. The approach maximizes the amount of static scheduling to reduce the need of context switch and operating system intervention. Experimental results show the potential of our method to reduce software design time and errors.


## 1 Introduction

We address the problem of optimal software synthesis for a set of concurrently communicating sequential processes to be executed on a single processor.

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].

The synthesis process proposed in this paper consists of defining a set of tasks from the functional processes and generating a schedule for this set of tasks where the static component is maximized. Once the tasks have been generated, the actual compilation can be carried out with optimizations driven by the architecture of the processor used for the implementation.

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) [3, 4]. 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 [4]. Approaches that use variations of control-data flow graphs, proposed mainly in the context of high-level synthesis for hardware design [6, 2], 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 [7] and [12], which can be applied only to closed systems with single-rate communication. The work of [9] is related to ours, especially in the underlying Petri net model, but cannot handle multiple reads/writes from/to the same channel by a given process, nor synchronization-dependent control on multiple ports. Finally, the authors of [10] also use a Petri net-like representation and can handle data-dependent control, 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.

The paper is organized as follows: in Section 2, we give an overview of our approach with an example. In Section 3, the Petri net model is presented and Section 4 describes the scheduling algorithm and the code generation process. In Section 5, the implementation of the overall approach and some experimental results are offered.

## 2 Overview

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. Multi-rate communication is supported, i.e. the number of objects read or written by a process at any given time may be an arbitrary constant. The system communicates with the environment 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. Controllable ports are under the system's control, i.e. reading data from them can be performed at any time. Uncontrollable ports are under the environment's control and the system must be ready to receive objects from them and react accordingly by performing operations. Without loss of generality, we will assume all primary input ports to be uncontrollable, since controllable ports do not impose any constraint on the system's operations.

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 called *FlowC*, which is based on C and extended in order to specify communication operations.

A task is generated for each uncontrollable input port, which performs the operations required to react to an event of that port. The code of the tasks will then be compiled and optimized for a particular architecture. Thus it is important to generate

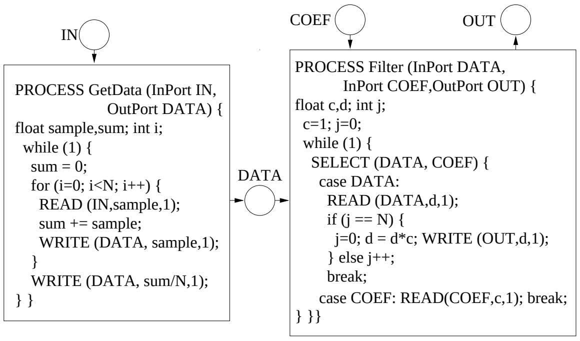


Figure 1: System specification

them so that the compiler can take full advantages of its optimization power to realize high quality implementation. Our approach can be considered as an architectural-independent preprocess step that globally analyzes all possible execution flows over the processes. It generates large blocks of code in which several processes may be interacting according to the particular reaction to one input port. It thus provides a better starting point for architectural-dependent optimizations, such as parallelization for VLIW processors, than the original specification.

Finally, all the tasks share the same memory space, thus minimizing context switching costs. Given that the execution of each task is driven by the occurrence of environment events, the intervention of the operating system is drastically reduced.

**Synthesis.** Our algorithm generates a sequential program for each task. All the possible execution flows of the tasks are represented by a *schedule* that has the property that the tasks can be executed with finite memory for arbitrary input streams, under the assumption that the system is fast enough to serve all environmental events. A schedule obtained by first finding operations specified in various concurrent processes that need to be executed when the ports receive inputs from the environment, and sequentializing them, while ensuring their execution with finite memory for the communication channels. The resulting schedule, if 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, one that guarantees the execution within the bound is sought. The specification may contain data-dependent control constructs, such as `if-then-else` or `for` loops, 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 schedule is computed as a directed graph annotated with objects of the Petri net. It is then transformed into tasks by traversing the graph to annotate software code.

**An example.** Figure 1 depicts the specification of a concurrent system with two processes, two input ports (`IN` and `COEF`) and one output port (`OUT`). The processes communicate to each other through the channel `DATA`.

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

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

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

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

Figure 2: Event-driven tasks after code generation.

This example also illustrates the main extensions of C language to support communication. The operations to communicate through ports have syntax `READ_DATA (port, data, nitems)` and `WRITE_DATA (port, data, nitems)`. The parameter `nitems` indicates the number of objects involved in the communication. This allows to support multi-rating, although the example uses only 1-object read/write operations. 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 when the number of items in the channel would exceed some pre-defined bound after writing.

The `SELECT` statement supports synchronization-dependent control, which specifies control depending on the availability of objects on input ports. In the example, the `SELECT` statement in `Filter` non-deterministically selects one of the ports with available objects. In case none of them has available objects, the process blocks until some is available. The `SELECT` statement is also extended to output ports (for the availability of space to store data) and to multi-rating (for the availability of more than one object). `SELECT` is a crucial statement to model reactive systems with several input ports, where the system is often waiting for the occurrence of events at any of the ports and reacts by non-deterministically choosing one of them.

Figure 2 shows the synthesized software after applying the method proposed in this paper. The task `Init` is executed for the initialization of the system. After that, the tasks `Tin` and `Tcoef` are scheduled upon the occurrence of an event at the ports `IN` and `COEF` respectively. The intervention of the operating system is negligible if, for example, an interrupt-based mechanism is used to wake up the tasks. All variables are global and shared by all tasks. Given that all possible execution flows of the tasks are known at compile time, a global inter-task data-flow analysis can be performed. The resulting code is much more adequate for the types of optimizations performed by compilers. For example, the variable `DATA`, that holds the objects sent through the channel, can be eliminated by simple optimizations such as *copy propagation* and *dead-code elimination* [1].

## 3 From specification to Petri net

In this section the translation from the specification of the system into a Petri is described. First of all, some basic notions on Petri nets are presented.

### 3.1 Petri Nets

A Petri net 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 another function from $P$ to non-negative integers, where $M[p]$ denotes the number of *tokens* at $p$ in $M$. $M_0$ is 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. 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 $P$.

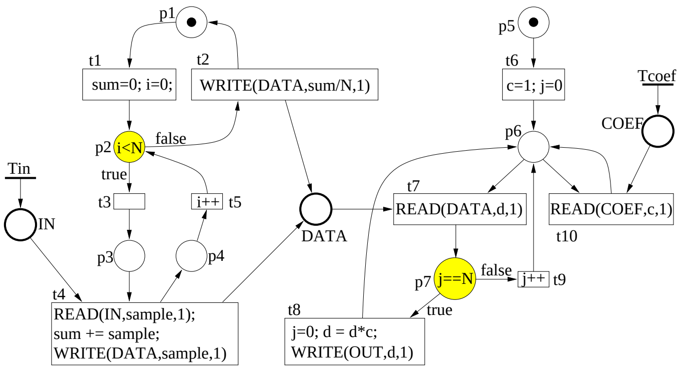


Figure 3: Petri net obtained from the specification of Figure 1. Places and transitions are represented by circles and rectangles respectively, where associated code is shown for each transition.

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 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 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 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 transitions, and each equivalence class is called *equal conflict set* (ECS). Note that the set of source transitions is an ECS, and we denote it by $E_U$. 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 [8]) if all the successor transitions are in the same ECS.

### 3.2 Translation into Petri net

The network of processes is transformed into a single Petri net, which is built in two steps: compilation and linking.

In 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 port operation is present in its body.

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.

If we ignore the places associated to the ports, the Petri net of one process obtained by the compilation strategy mentioned above has exactly one place marked at each reachable marking. 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.

Linking combines Petri nets generated in compilation into one, by merging each pair of places for ports connected by a channel. If a bound is defined for a channel, it is represented as 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. Controllable input ports need not to be modeled with an explicit place and source transition.

Figure 3 depicts the Petri net obtained from the specification of Figure 1.

## 4 Software Synthesis

In this section, the two basic parts of our software synthesis approach are presented: scheduling and code generation. We first present a formal definition of a schedule.

### 4.1 Definition of Schedules

A *schedule* for a given Petri net 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 five properties. First, there exists exactly one node associated with the initial marking. Second, for each node $v$, the set of transitions associated with the edges out of $v$ is an ECS enabled at $M(v)$. If this ECS is the set of source transitions, $v$ is called *await node*. Third, for each edge $[v, w]$, $M(v) \xrightarrow{T([v,w])} M(w)$ holds. Fourth, each node has at least one path to an await node. Fifth, each await node is on at least one cycle.

Intuitively, scheduling can be deemed as a game between a scheduler and the environment. Starting from the node associated with the initial marking, the scheduler traverses the schedule, firing transitions of the visited edges. When it reaches an await node, it ceases the traversal, waiting for the environment to fire a source 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 fourth and fifth properties guarantee that at any moment of the traversal, there is a path to an await node that is on a cycle. This ensures that the scheduler can always proceed to a state at which the environment may fire transitions, and cyclic behavior is established from the state.

### 4.2 Scheduling

The scheduling algorithm creates dynamically a subtree of the reachability tree. The created tree is composed by nodes that represent the system states and arcs that represent transitions which produce new states. A post-processing creates a cycle for each leaf to generate a schedule. Initially, we create the root $r$ and set $M(r)$ to the initial marking of the Petri net. The algorithm then calls a function $\texttt{EP}(r, r)$, shown in Figure 4.

`EP` takes as input a leaf $v$ of the current tree and its ancestor *target*. We say that a node $u$ is an *ancestor* of $v$, denoted by $u \leq v$, if $u$ is on the path from the root to $v$. If in addition

```
function EP(v, target)
    EP ← UNDEF, ECS(v) ← φ;
    if(termination conditions hold) return (0,UNDEF);
    if(∃u : u < v and M(u) = M(v)) return (0, u);
    for(each ECS E enabled at M(v))
        if(E = E_U) current_target ← v;
        else current_target ← target;
        (AF_ECS, EP_ECS) ← EP_ECS(E, v, current_target);
        if(AF_ECS = 1)
            ECS(v) ← E, return (1, EP_ECS);
        if(EP_ECS ≤ current_target)
            ECS(v) ← E, return (0, EP_ECS);
        if(EP = UNDEF or EP_ECS < EP)
            ECS(v) ← E, EP ← EP_ECS;
    return (0, EP);
```

(a)

```
function EP_ECS(E, v, target)
    AF_ECS ← 0, 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);
        (AF, EP) ← EP(w, current_target);
        if(AF = 1 or ECS(w) = E_U) AF_ECS ← 1, current_target ← v;
        else if(EP = UNDEF or v < EP) return (0, UNDEF);
        EP_ECS ← min(EP_ECS, EP);
        if(EP_ECS ≤ target) current_target ← v;
    return (AF_ECS, EP_ECS);
```

(b)

Figure 4: Various termination conditions can be adopted in EP. An example is bounds on the places of the Petri net, i.e. a positive integer called *bound* may be associated with a place, and the termination condition holds if $M(v)[p]$ exceeds the bound of some place $p$. Other useful conditions are found in [11]. $E_U$ denotes the set of source transitions. In the function EP_ECS, $T([v,w])$ denotes the transition associated with the edge $[v,w]$. $AF\_ECS$ is a boolean variable that is set to 1 if $v$ has a path to an await node.

$u \neq v$, $u$ is a *proper ancestor* of $v$, denoted by $u < v$. EP creates a tree rooted at $v$, so that a path can be created from $v$ to some await node in the resulting schedule. It also makes sure that all the await nodes in the tree will be contained in cycles when generated in the post-processing. Specifically, EP returns two values. The first is a boolean variable, which is 1 if and only if $v$ has a path to an await node in the created tree. The second is an ancestor $u$ of $v$, for which the marking $M(v)$ satisfies the following: 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. $EP$ tries to find such a $u$ by calling a function EP_ECS for each ECS enabled at $M(v)$. If there is such a $u$ with $u \leq target$, EP returns one. Otherwise, it returns such a $u$ closest to the root, if exists. In either case, $ECS(v)$ is set to the ECS with which the property holds for $u$ and $M(v)$. If no ancestor $u$ with this property exists, a special value UNDEF is returned.

The ECS's enabled at $M(v)$ are sorted in the function EP, where the set $E_U$ of source transitions is positioned at the end of the order, and EP_ECS is called in this order. This minimizes the number of await nodes introduced in a schedule. Denoting by $AF$ and $EP$ the two values returned by EP$(r, r)$, we say that the algorithm succeeds if either $AF = 1$, or else $EP = r$ and $ECS(r) = E_U$. This means either $r$ has a path to an await node for which an ancestor $u$ has been found with the property above, or $r$ is an await node and such an ancestor $u$ is $r$ itself. In this case, we call the post-processing to create a schedule and terminate. Otherwise, we report no schedule and terminate.

The post-processing consists of two parts. First, we retain only a part of the created tree that are used in the resulting schedule, and delete the rest. The root is retained, and a node $w$ is retained if its parent $v$ is retained and the transition $T([v,w])$ is in $ECS(v)$ set by EP [1]. Second, a cycle is created for each leaf $w$ of the retained portion of the tree, by merging $w$ with its proper ancestor $u$ such that $M(u) = M(w)$. By construction, such a $u$ uniquely exists for $w$. The graph obtained at the end is returned. It is shown that this algorithm always finds a schedule, if there exists one in the space defined by the terminate conditions employed in EP [11]. See Figure 5 for an example.

When ECS's are sorted in EP, those other than $E_U$ are ordered using *t-invariants*. This heuristic helps finding a schedule sooner, and often keeps the resulting graph small. It also checks a sufficient condition under which a schedule does not exist. Thus if the condition is met, we can terminate immediately reporting no schedule. We briefly illustrate the heuristic in the rest of this section.

A t-invariant is a vector of non-negative integers that solves equations $Cx = 0$, where $C$ is a $|P| \times |T|$ matrix given by $C_{ij} = F(t_j, p_i) - F(p_i, t_j)$. It 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. If such a sequence can be fired from a marking $M$, the marking obtained after the firing is also $M$. Our heuristic computes a t-invariant, and EP processes ECS's starting from those that have at least one transition in the sequences represented by the vector. The vector is updated by decrementing integers whenever transitions are processed in EP_ECS, and if the result is $-1$, we compute a new t-invariant.

A known problem with t-invariants is that it is in general difficult to find if a t-invariant has a sequence that can be fired at a given marking [8]. In our case, however, a necessary condition can be obtained due to the structure of a Petri net generated from a *FlowC* specification. The *FlowC* compiler translates a process in the network to a Petri net so that exactly one place has a token in any reachable marking $M$, as described in Section 3. Let us refer to as a pseudo-enabled ECS at $M$ the ECS that the successors of this place belongs to. If a t-invariant is fireable at $M$, it is necessary that for each process, either its pseudo-enabled ECS has a transition in the invariant, or no transition that belongs to the process appears in the invariant. Therefore, we first compute a basis of non-negative integers for the equations $Cx = 0$, and then find a subset of the basis so that the sum of the vectors of the subset satisfies this necessary condition. This problem can be formulated as a covering problem, and we heuristically find a subset with a minimal cardinality [2]. If no basis is found for the equations, it is known that no schedule exists, and we terminate the procedure immediately.

### 4.3 Code Generation

The code generation algorithm takes a schedule $S$ and synthesizes code. A direct translation of the schedule into code is possible but usually increases the code size, since different paths of the schedule may be associated with the same sequence of transitions, which yields a same code segment. We thus perform optimizations to minimize the code size.

[1] In the actual implementation, this process is done dynamically in EP.

[2] If no such subset exists, we do not sort ECS's in the current EP, and re-compute a new t-invariant when EP is called next.

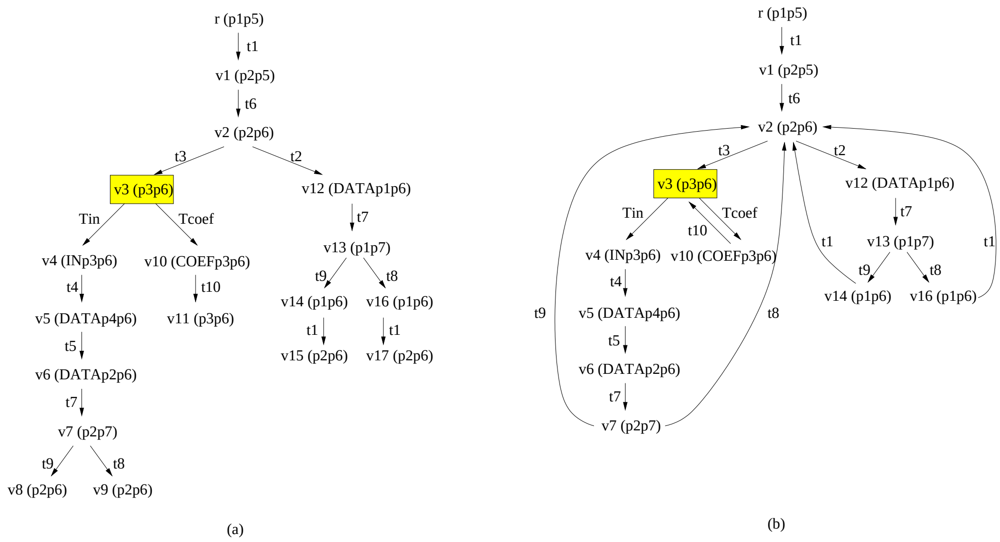


Figure 5: (a) shows the tree obtained just before the post-processing, when the algorithm is applied to the Petri net of Figure 3. Suppose that we use bounds of places as the termination condition, where we set a bound for each place equal to 1. The marking associated with each node is shown in parentheses adjacent to the name of the node. (b) presents the final schedule. At each node we assume that the ECS shown is processed first, among those enabled at the marking. Suppose that the procedure has arrived at $v_2$. `EP`($v_2$, $r$) is called at this node, which then calls `EP_ECS`($\{t_3, t_2\}$, $v_2$, $r$). `EP_ECS` creates a node $v_3$ for a transition $t_3$ and calls `EP`($v_3$, $r$). The only ECS enabled at the marking of $v_3$ is the set $E_U$ of source transitions. Thus `EP` sets *current_target* to $v_3$ and calls `EP_ECS`($E_U$, $v_3$, $v_3$). `EP_ECS` then processes each of the two source transitions. Consider the transition $T_{in}$, and suppose that the procedure has arrived at $v_7$. The *target* is still $v_3$, and thus `EP`($v_7$, $v_3$) is applied. `EP` calls `EP_ECS` with the ECS $\{t_8, t_9\}$. `EP_ECS` then creates a node $v_8$ for the transition $t_9$ and calls `EP`($v_8$, $v_3$). Since the marking of $v_8$ is equal to that of $v_2$, `EP` returns (0, $v_2$), which makes `EP_ECS` set $EP\_ECS$ to $v_2$. Since $EP\_ECS \leq target$ holds, i.e. $v_2$ is an ancestor of the target $v_3$, `EP_ECS` sets *current_target* to $v_7$. It then processes the other transition $t_8$, for which `EP` returns (0, $v_2$). Suppose now that the procedure has come back to the node $v_3$, at which `EP_ECS`($E_U$, $v_3$, $v_3$) returns (0, $v_2$). Since *current_target* had been set to $v_3$ in `EP`($v_3$, $r$), $v_2 \leq current\_target$ holds. Therefore, `EP`($v_3$, $r$) immediately returns (0, $v_2$) to `EP_ECS`($\{t_2, t_3\}$, $v_2$, $r$). $ECS(v_3)$, the ECS assigned at $v_3$, is $E_U$, and therefore `EP_ECS`($\{t_2, t_3\}$, $v_2$, $r$) sets $AF\_ECS$ to 1 and *current_target* to $v_2$. It then continues for the transition $t_2$ by calling `EP`($v_{12}$, $v_2$). It will return (0, $v_2$), and `EP_ECS`($\{t_2, t_3\}$, $v_2$, $r$) returns (1, $v_2$). These values are propagated to the root, and are finally returned by `EP`($r$, $r$). The post-processing is then called, which deletes the nodes $v_8$, $v_9$, $v_{11}$, $v_{15}$, and $v_{17}$, and creates cycles as shown in (b).

The algorithm is in three steps. First, we traverse the schedule to identify code segments. 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 $ECS(v')$. The traversal generates a minimal set of code segments with the property that for each node $v$ of $S$, the set has exactly one node $v'$ with $ECS(v) = ECS(v')$. We say that such a $v$ corresponds to $v'$. Further, each leaf $v'$ of a code segment maintains a set of pairs made of a marking and an ECS defined as follows. For each node $u$ of $S$ that corresponds to the parent $u'$ of $v'$, $(M(v), ECS(v))$ is included in the set, where $v$ is the child of $u$ such that the transition $T([u, v])$ is associated with $[u', v']$. This set is used in the second step.

The second step generates a function code in which the generated code segments are translated. At the beginning, we define global variables for the places, which are used to represent markings, and are shared among all tasks. They are initialized to the initial marking $M_0$, and whenever a transition is translated, we increment or decrement the variables of the places whose token counts change by firing the transition. Then, we translate the segments, starting from the one whose root has the ECS equal to $E_U$: for each source, a function is generated with its name, so that it's to invoke the corresponding task. Other segments within a task may be translated in any order. For a given segment, we first create a label whose name is derived from the ECS of the root of the segment, and then visit each node in a depth-first manner. For a non-leaf node $v'$, the original FlowC code is copied for each transition of $ECS(v')$. If $ECS(v')$ has more than one transition, an `if-then-else` construct is generated in addition, with the condition taken from the FlowC code. For a leaf $v'$, code is generated so that the execution jumps to the code segment to be executed next. This code segment depends upon the current marking given by the global variables. We generate a `switch` construct, where for each pair $(M, ECS)$ in the set associated with $v'$, a `case` is generated so that if the current marking is equal to $M$, a `goto` jumps to the label created for the $ECS$. By the property of code segments given in the first step, such a label uniquely exists. If the $ECS$ is $E_U$, we generate a `return` instead of a `goto`, since we cannot continue the execution until further inputs are received.

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

Figure 6 explains more in details how the code segments are generated, and Figure 2 shows the synthesized code for the same example. In this case, additional optimizations have also been applied: unrolling eliminates some `goto`s and global variables for places that are not used.

## 5 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.

The system we used is made of four processes, and is schematically depicted in Figure 7. 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 `start`. All the process are potentially concurrent and communicate through FIFO channels. The producer, filter, consumer chain constitutes the *hard real-time* video data

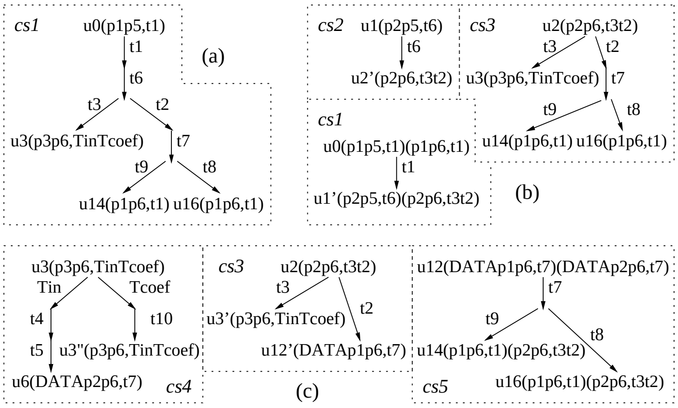


Figure 6: The code generation algorithm is illustrated for the schedule of Figure 5-(b). The five resulting code segments are shown in (b) (only $cs_1$ and $cs_2$) and in (c) (segments $cs_3$, $cs_4$ and $cs_5$). The algorithm recursively traverse the schedule to identify the code segments, stopping at each *await node* or when a transition which is already in a code segment is found: the first step creates a code segment $cs_1$ shown in (a), where the pairs of a marking and an ECS are indicated in parenthesis for the root and the leaf nodes. It stops at node $u_3$ as the corresponding node $v_3$ in the schedule is an await node, and at nodes $u_{14}$ and $u_{16}$ because the outgoing transition from $v_{14}$ and $v_{16}$ is $t_1$, already present in $cs_1$. The second step starts from $v_{14}$: it immediately recognize that $cs_1$ should be split, because from $t_1$ you can either go to $t_6$, or to the choice between $t_3$ and $t_2$; therefore we get three code segments, as shown in (b). The third step starts from $v_{16}$ and does not need to create any new code segment. The fourth starts from $v_3$ and creates a new code segment $cs_4$ rooted at $u_3$: it stops at $u_3''$ (an await node is reached) and at $u_6$; the next transition after $u_6$ is $t_7$, already present in $cs_3$ which thus needs to be split, generating a new $cs_3$ and $cs_5$, as shown finally in (c).

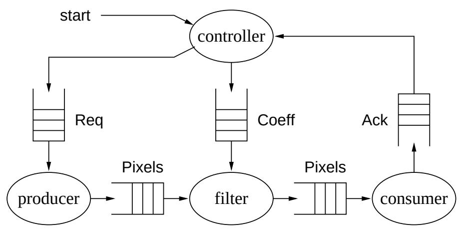


Figure 7: Process network for a video application

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.

Our proposed algorithm generated, in less than a minute, a single task with all the channels of size 1. This has been compiled and profiled on a MIPS R3000 machine, and was compared to the case where the original four processes were implemented as separate tasks and executed by a round-robin scheduler.

We performed several experiments varying the size of the channels and the number of frames transmitted. 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 8. The $y$ axis reports the number of clock cycles and the $x$ axis the size of the channel buffers. 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). 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

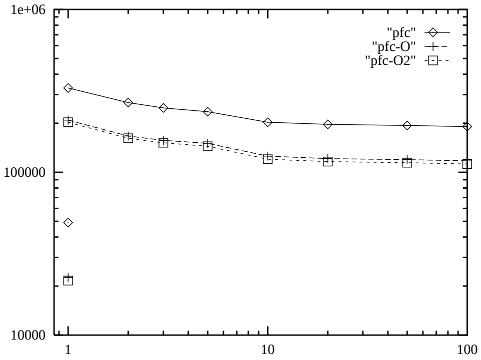


Figure 8: Execution time and memory requirements for single task versus four tasks (with various buffer sizes).

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.

## 6 Conclusions

Given a set of concurrent processes described with a "high-level" representation, a procedure to generate a set of tasks from the processes as well as a schedule for their execution on a single processor have been presented. The method is based on Petri nets and can be fairly easily implemented in a flow that can substantially reduce software design time and errors.

**Acknowledgment**

The authors are grateful to J-Y. Brunel, A. Kenter, E. de Kock, W. Kruytzer, and W. Smits of Philips Research Labs for their inputs and interaction throughout this work, especially on the semantics of *FlowC* specifications and on experiments. This work is supported by ESPRIT-COSY EP25443, CICYT TIC 98-0410 and 98-0949, the CNR and GRQ 1999SGR-150.

## References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] S. Amellal and B. Kaminska. Functional synthesis of digital systems with TASS. *IEEE Trans. CAD*, 13(5), 1994.

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

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

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

[6] G. Lakshminarayana, K. Khouri, and N. Jha. Wavehead: A novel scheduling technique for control-flow intensive designs. *IEEE Trans. CAD*, 18(5), 1999.

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

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

[9] 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.

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

[11] *the same authors.* Task generation and compile-time scheduling for mixed data-control embedded software. Technical Report LSI-99-47-R, Dept. of Software, Universitat Politècnica de Catalunya, 1999.

[12] F. Thoen, M. Cornero, G. Goossens, and H De Man. Real-time multitasking in software synthesis for information processing systems. In *International System Synthesis Symposium*, 1995.