From the handouts of the Second International Workshop on Integration of Specification Techniques for Applications in Engineering (INT'02)

Synthesis of Embedded Software for Reactive Systems

Robert Clarisó[1] Jordi Cortadella[1] Alex Kondratyev[2]
Luciano Lavagno[3] Claudio Passerone[3] Yosinori Watanabe[2]

[1] Universitat Politècnica de Catalunya, Barcelona, Spain *
[2] Cadence Berkeley Labs, Berkeley, USA
[3] Politecnico di Torino, Torino, Italy

Abstract: Embedded systems are usually deployed in environments with tight timing constraints. Increasing run-time performance is often a critical issue. This paper addresses the problem of synthesizing efficient software for embedded systems specified as a set of communicicating sequential processes. The Quasi-Static Scheduling algorithm is presented, along with results from an industrial example. Open problems in this field, such as incorporating data-flow analysis into the scheduling algorithm, are discussed.

1 Introduction

1.1 Embedded systems

Embedded systems are driving research from the electronic community due to their widespread use. In our daily round we interact with multiple embedded systems: a VCR, a cellular phone, an electronic organizer, a digital camera, etc. The design of such devices requires a multidisciplinar knowledge of software and hardware systems. It also requires a precise knowledge of how the system interacts with the environment.

More formally, an *embedded system* is a specialized computation unit that performs a set of tasks within a larger system called *environment*. The interaction with the environment is typically *reactive*, i.e. the embedded system must process inputs from the environment under some timing constraints such as the speed (throughput) or the delay (latency).

*This work has been partially funded by CICYT 98-0410 and by a grant from Cadence Design Systems

As embedded systems are highly concurrent by nature, specifications often employ formal models [6, 5] that expose parallelism explicitly. Some examples of formal models are data flow networks [15], Petri Net based models [7], Communicating Sequential Processes [13] and Kahn process networks [14].

The implementation of these specifications is generally a software component running on top of a hardware architecture that might include several CPUs, DSPs, co-processors, and so on. The synthesis of this software-hardware mixed implementation defines three problems: (a) a *mapping* from elements of the specification to hardware and software components, (b) an *allocation* of software components to hardware computation units (e.g. CPUs) of the embedded system, and (c) a *scheduling* of the software components in each of the hardware computation units. Technology mapping and allocation are dependent on several factors, including cost and technology constraints. However, the remaining scheduling problem allows a formal approach as it is only dependent on the constraints dictated by the environment.

1.2 Scheduling of reactive systems

In the case of reactive systems, the scheduling problem is subject to timing constraints on the response to the inputs from the environment. Minimizing run-time of the input processing, if possible, and minimizing idle CPU time will the main concerns of the scheduling algorithm. The former is a matter of choosing the best algorithm and using an optimizing compiler. But the latter can be improved by reducing the CPU time spent by the operating system.

One possible scheduling approach could be *dynamic scheduling*: we generate one process for each functional process in our specification, letting the operating system decide which process should be executed at run-time. The problem with dynamic scheduling is that task context switching causes a run-time overhead, which can be unacceptable in some environments. Therefore, although dynamic scheduling is used widespread in interactive systems, it is not appropiate for reactive systems.

Static scheduling techniques try to make most decisions about execution order at compile-time, thus reducing the time consumed by the operating system during run-time. These algorithms require that some information about the rate of arrivals of inputs from the environment is known at compile-time. Even then, completely static schedules are only possible if we restrict ourselves to specifications without data-dependent choices. Real-time embedded systems, where a failure to satisfy a deadline can lead to a serious failure, can cope with this restrictions. But generic embedded systems would be very limited without data-dependent constructs. The problem of finding partial schedules in a concurrent system with data dependent constructs ensuring that there will not be buffer overflow is known as *Quasi*

Static Scheduling (QSS).

This paper presents the QSS problem and a static method to synthesize a set of tasks for a concurrent system that provide better run-time performance that a set of processes with run-time scheduling. The rest of sections are organized as follows. Section 2 provides theoretic background and necessary definitions from the PN domain. Section 3 contains a high-level description of the algorithm, and section 4 presents some results obtained from an industrial example. Section 5 describes relevant open problems that will probably lead to future research on the area.

1.3 A motivating example

Figure 1 shows an example of embedded system that controls the environmental conditions in a room of a library. The books in the room can be damaged by a high humidity or a quick rise in temperature. Two sensors in the system monitor the temperature and humidity in the room; the system can react by activating air conditioning or a dryer. Finally, if the temperature becomes too high, the system assumes that there is a fire and raises an alarm. The system is divided in three processes. Two processes, namely the `TEMP-FILTER` and `HUMIDITY-FILTER`, filter the readings from the sensors that are not relevant to the system, such as low humidity or small change in temperature. A third process, the `CONTROLLER`, chooses the correct reaction to the current condition of the system. The communication among the three processes is performed using two internal ports, `TDATA` and `HDATA`.

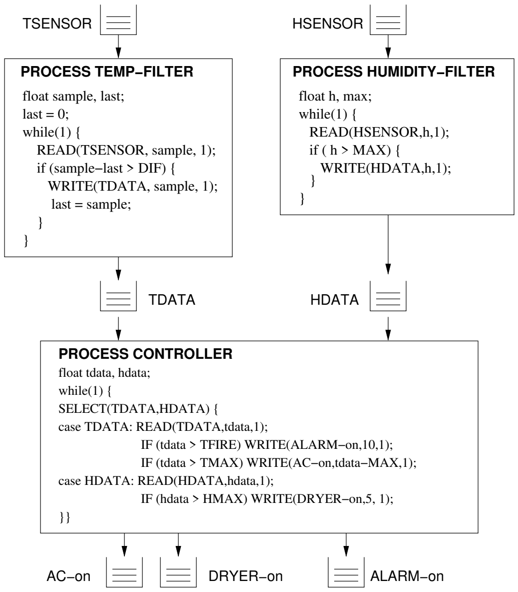
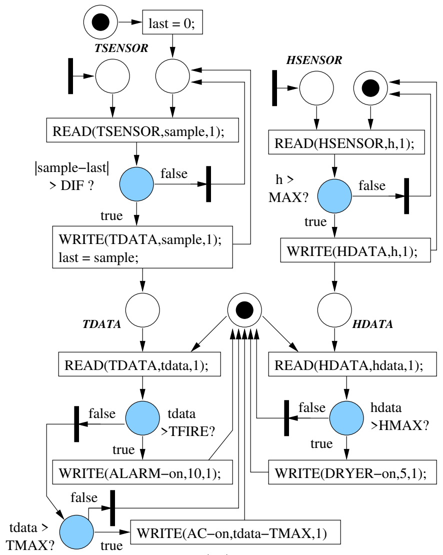


Figure 1: System specification example: environmental controller



(a)

```
Init() {
  last = 0;
}
```

```
Hsensor() {
  READ(HSENSOR, h, 1);
  if (h > MAX)
    if (h > HMAX)
      WRITE(DRYER−on, 5, 1);
}
```

```
Tsensor() {
  READ(TSENSOR, sample, 1);
  if (sample − last > DIF) {
    last = sample;
    if (sample > TFIRE) {
      WRITE(ALARM−on, 10, 1);
    } else if (sample > TMAX)
      WRITE(AC−on, 5, 1);
  }
}
```

(b)

Figure 2: (a) Flow C specification, (b) Event-driven tasks after code generation

The system is described using a language called *FlowC*. FlowC is basically a C language augmented with communication and synchronization constructs. The constructs related with communication specify the port where the information is read/written, the values read or written in these ports and the number of elements read or written: `READ(port, item, nitems)` and `WRITE(port, item, nitems)`. Communication has blocking semantics and it uses a FIFO policy. The third argument of `READ` and `WRITE` allows multi-rate communication among processes. Several `READ` (`WRITE`) constructs can use different values for `nitems` as long as the value is a constant known at compile-time. Synchronization is provided by means of a `SELECT` construct, that is used when the system is waiting for an input in a set of ports. The semantics is that when there is an input in a port or ports of the set, `SELECT` non deterministically chooses one of these available input ports.

A classical implementation would require three processes and an operating system that schedules their execution at run-time. We will analyze the results obtained from QSS and compare them with this implementation.

The QSS algorithm will generate a task for each input from the environment, i.e. there will be one task to process a temperature input and another one to process a humidity input. These tasks will be generated in three steps. First, a Petri Net model of the system will be built from the FlowC specification. Then, the Petri Net will be analyzed to find a set of run-time schedules that cover the run-time behavior of the system. Finally, we will generate code for these schedules.

Figure 2(a) shows the Petri Net generated by QSS from the environmen-

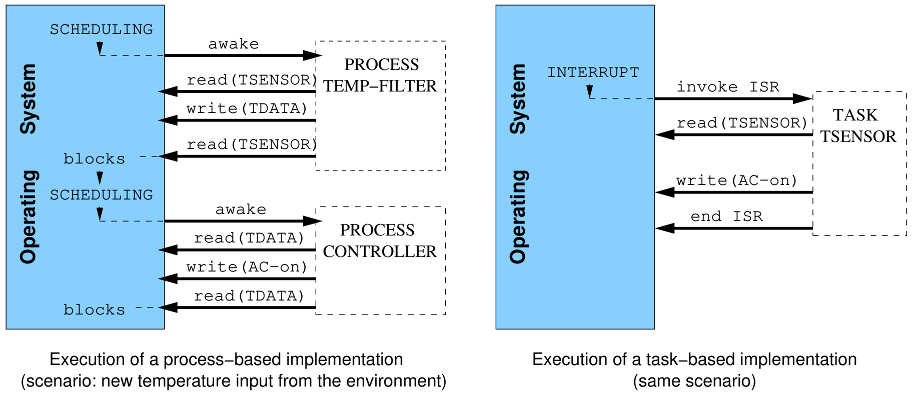


Figure 3: Comparison of task-based implementation versus process-based implementation.

tal controller example. The code generated for the tasks in the system is displayed in 2(b). There is task for initialization, but it is negligible as it only has to be performed once at start-up. It is important to notice that, even though there are three processes in the original specification, there are only two tasks in the result of QSS, one for each input. The intuition for this is that each task performs the maximum amount of processing for the available input; although this sometimes implies a code replication among tasks, it reduces the amount of inter-task communication, because two tasks won't communicate unless two inputs are required to perform a computation.

The main advantage of a task-based implementation versus a process-based implementation is the reduction of the intervention of the operating system. A task-based implementation can use an interrupt based mechanism to wake up tasks when the system receives an input, which is much faster than the context switching required by a scheduling algorithm. The consequence is more time devoted to the execution of the system code and less time executing code from the operating system.

Figure 3 compares the process-based and task-based implementations. Task based implementation requires less interaction with the operating system, because of two reasons. The first one is that scheduling is not required. The second one is that processes require more internal communication to complete the task. Both factors justify our claims that a task-based implementation provides a much better performance than a process-based implementation.

1.4 Benefits and problems of the QSS approach

The QSS algorithm provides a mechanism to synthesize a task-based implementation from a concurrent specification. This implementation provides a better performance than a task-based implementation due to three factors:

- Reduction of the intervention of the operating system during run-time. In particular, the cost of run-time scheduling is reduced greatly as task context switching is not used if *all tasks use the same memory space* and *an interrupt-based mechanism is used to wake up the tasks.*
- Reduction of the memory requirements for our system. Communication buffers use up memory space, and they are required for inter-process communication. A task-based implementation implies less communication inside the system, so several of these buffers can be removed. Writing/reading from a buffer can be replaced by writing/reading from a scalar. For example, in Figure 2 channels `HDATA` and `TDATA` have been replaced by scalar variables.
- Improved chances of a good code optimization. As QSS replicates some code in several tasks, each task contains all the code related to the response to a particular input. Therefore, the compiler has new opportunities to perform code optimizations on the tasks, that were not available before due to the separation of processes. Moreover, replacing inter-process communication by scalar reads/writes makes the resulting code very adequate for data-flow analysis and optimization. Simple techniques like *copy propagation* and *dead-code elimination* [1] can noticeably improve the performance of the code of the tasks.

There are two main problems with the QSS approach.

- Replication of code among different tasks. This was considered a benefit, as it enabled further optimizations. However, it also has its drawback: the size of the code increases and therefore more memory is needed to store the code for the system. This problem is not so relevant, as time usually is much more critical than memory and reduction in memory usage by eliminating buffers makes up for this increase.
- Non-schedulability of some systems. The QSS problem has been proven to be undecidable. Our method copes with undecidability by accepting failures to generate schedules for some systems; scheduling these systems might be possible by taking into account additional information that renders the problem undecidable. Finding a mechanism to schedule these systems is currently an open problem. Refer to section 5 for more discussion on this topic.

2 Petri Nets

This section describes general notions of Petri Nets used in the description of the algorithm. Further theory, definitions and properties of Petri Nets can be found in [2].

A Petri Net is a 4-tuple $PN = (P, T, F, M_0)$ where $P = p_1, \ldots, p_n$ is a finite set of *places*, $T = t_1, \ldots, t_m$ is a finite set of *transitions*, $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the *flow relation* and $M_0 : P \to \mathbb{N}$ is the *initial marking*. This tuple 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. Transitions are tipically drawn as bars or boxes, while places are represented by circles.

Each place on the Petri Net contains a number of *tokens*, defined by a *marking* $M : P \to \mathbb{N}$. The number of tokens in a place p at a marking M is denoted by $M[p]$. The initial marking describes the initial state of the places of the net. A marking might enable a transition, meaning that this transition can be fired; firing an enabled transition modifies the marking, possibly enabling or disabling other transitions. Formally, a transition t is said to be *enabled* at a marking M if $M[p] \geq F(p, t)$ for all $p \in P$. 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 $p \in P$. In the sequel, $M[t\rangle M'$ denotes the fact that marking M' is reached from marking M by firing the enabled transition t.

A transition is called a *source* transition when $F(p, t) = 0$ for all $p \in P$. Source transitions are always enabled independently of the current marking, and therefore they can always be fired at any time. By this reason, these transitions will be used to model input events from the environment.

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 reachable markings from the initial marking is denoted by $[M_0\rangle$. The *reachability tree* of a Petri Net is a tree that contains nodes labeled with the marking in $[M_0\rangle$; the root node is labeled with M_0 and a node labeled with M has a child M' if there is a transition t such that $M[t\rangle M'$. 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 to define 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 $p \in P$. These transitions are in conflict in the sense that t_i is enabled at a given marking only if t_j is enabled, i.e. if the firing of a transition disables t_i it also disables t_j. The equal conflict is an equivalence relation on the set of transitions, that can be partitioned in set of equivalent transitions called *equal conflict sets* (ECS). For example, the set of source transitions is an ECS, which is denoted 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 its has more than one successor transition. A choice place is *Equal Choice* if all the successor transitions are in the same ECS.

3 Software synthesis algorithm

3.1 Assumptions

The QSS algorithm presented on this paper works under a set of assumptions on the environment and the system. These assumptions must hold in order for the QSS algorithm to be appliable.

We consider a system to be specified as a set of concurrent processes. The implementation of this system is mapped as software to be executed on a single programmable processor.

The system interacts with the environment through *input* and *output* ports. The interaction proceeds as follows: the environment can place an object in an input port at any time; the system reacts to this input by performing some actions, which may result in writing an object to an output port; the environment can consume the objects in output ports at any time.

The only assumption made on the inputs is about the *rate*: the rate of the arrivals of inputs cannot be faster than the rate of processing of inputs. This assumption is necessary because otherwise it would be impossible to avoid overflow. No other assumptions are made on the inputs, in particular we assume that there is no correlation among inputs.

The behavior of each process in the system is described by a sequential program, written in a language called FlowC. Communication among processes is point-to-point and occurs through uni-directional FIFO ports.

3.2 Overview of the QSS algorithm

Under these assumptions, the QSS algorithm will generate a sequential task for each input port. The tasks are guaranteed to require a finite memory for the communication ports, as the QSS algorithm computes a bound to the maximum number of elements in each port of the system. If the algorithm fails to establish bound for any port, QSS fails and the system is considered to be *non-schedulable*. The synthesis of tasks is performed using Petri Nets as the underlying model, due to their explicit representation of concurrency.

The code of each task might contain data-dependent control constructs, like conditionals or loops, because they cannot be resolved without the values of data, that are only available at run-time. All other operations, which are not data-dependent, are sequencialized to reduce the run-time overhead.

Before introducing the QSS algorithm in detail, we will present a very brief overview of the three main steps of the algorithm:

$$\text{Comm. sequential processes} \longrightarrow \overset{\text{Quasi-Static Scheduling algorithm}}{\boxed{\xrightarrow{1} \text{Petri Net} \xrightarrow{2} \text{Schedules} \xrightarrow{3}}} \rightarrow \text{Tasks}$$

1. *Translation into Petri Net*: The concurrent specification is translated into a single Petri Net model. FlowC statements and data-dependent

constructs appear as labels in this Petri Net. Inputs from the environment are modeled as source transitions in the Petri Net.

2. *Scheduling*: The Petri Net is analyzed to find all the possible execution flows that can be taken when an input reaches the system. Those execution flows contain actions that belong to different processes and can be sequentialized. The result is an *schedule* for each input of the system.

3. *Code generation*: The last step consists in generating code for the schedules. The goal of this step is to reduce the size of generated code and to take advantage of bounds in communication buffers to increase performance, e.g. replacing buffers of size 1 by scalar variables.

3.3 Translation into Petri Net

The first part of the QSS algorithm is the translation of a set of concurrent processes into a single Petri Net. This translation is achieved in two steps: *compilation* and *linking*. Briefly, compilation generates a subnet for each process in the network, while linking combines those subnets into the final net.

Compilation performs a syntax-driven translation on the code of each process to get its subnet. A place is created for each port read or written from the process. The FlowC statements are translated differently depending on t: statements that do not operate with ports are labeled directly into a Petri Net, while statements that operate with ports are decomposed into several transitions until basic communication statements are reached. These basic communication statements (`SELECT`, `READ`, `WRITE`) are modeled as adding/removing tokens to/from a place related to a port.

When translating a compound statement (e.g. `while` loop) that contains a communication construct, data-dependent constructs are modeled as *Equal Choice* places, labeled with a boolean condition and with two outgoing arcs, labeled `True` and `False`. The succesor transitions of such a place constitute an ECS.

Petri net linking combines all the subnets from all the processes into a single subnet. This is done by (1) merging places that represent the same port into a single place and (2) adding one source (sink) transition connected to each input (output) port. This resulting Petri Net models the concurrency in the execution of the processes of the system. As stated previously, source transitions model inputs from the environment. Source transitions are always enabled, modeling that input transitions can be fired from the environment at any time. However, we assume that the input rate is low enough to allow the processing of inputs in the system.

3.4 Scheduling

When the Petri net model for the whole system is available, the next step is the generation of a set of traces that represent all possible behaviors at run-time. The evolution of a Petri Net can be modeled by means of the reachability tree, that specifies which markings are reachable from the initial marking (and hence the possible states of the communication channels), and which transitions can be fired (and hence, which code will be executed at each point). We do not need to explore the entire reachability space to model the behavior of the system, because we know that this behavior will be cyclic: *wait for input*, *receive input*, *react to input*, *wait for input again*. Therefore, the goal of this phase will be finding a relevant subset of the reachability space, i.e. a relevant subtree of the reachability tree, that is enough to model the cyclic behavior of the system regardless of the values of data, while ensuring finite memory for communication channels. This relevant subtree is called a *schedule*, and it provides a sequential summary of the response of all processes in the system to inputs from the environment. This schedule will be our first step towars a task-based implementation. From this schedule, we will describe the behavior of the system with respect to a single input, and from this model we will generate a task for each input of the system.

An important notion to be defined before describing schedules is the notion of *await state*. A state of the reachability tree where the only transitions that are enabled are the source transitions E_U is called an *await state*. As the only transitions that can be fired are inputs from the environment, an await state models a situation where the system is waiting for the environment to produce an input. Any run of a task of the system will start in an await state (the system was waiting and the environment produces an input) and will end in an await state (the input has been fully processed or it the system needs more inputs to continue its computation).

The definition of schedule is based on this notion of await state. A *schedule* of a PN is finite directed graph, where each node v is associated with a marking $M(v)$ and each edge (u, v) is associated with a transition $T(u, v)$. The graph has 5 properties:

1. There is only one node v associated with the initial marking M_0.

2. For each node v, the out edges are associated with transitions of an ECS that is enabled in marking $M(v)$. When this ECS is E_U, the source transitions, this is an await node. This ECS can map a data-dependent construct from the original specification.

3. For each edge (u, v), marking $M(v)$ can be reached from marking $M(u)$ firing transition $M(u, v)$.

4. Each node has at least one path to an await state.

5. Each await state is at least on one cycle

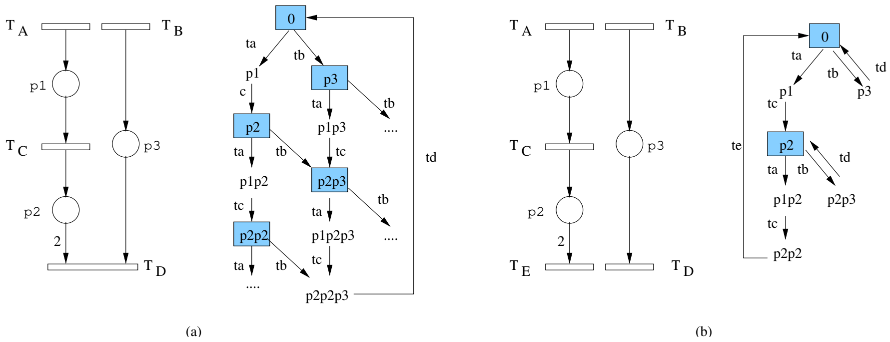


Figure 4: (a) Non-schedulable net, (b) Schedulable net with schedule

For example, Figure 4 presents the schedulability problem for two Petri Nets. For each PN, we try to build a schedule that ensures the five properties stated above. Await states in the schedules are displayed inside a grayed box. For net (a) we cannot find a finite schedule, thus the net is labeled as non-schedulable. The impossibility to schedule this net comes from the lack of control of the inputs from the environment `a` and `b`: one can be fired from the environment indefinitely, therefore we can reach a marking with an infinite number of tokens in the places `p2` or `p3`. However, we can find a finite schedule for net (b), as the reaction to one input is independent from the other inputs from the environment.

A complete definition of the scheduling algorithm can be found in [7]. Intuitively, schedules are built as subtrees of the reachability tree. The traversal starts at the initial marking, and it explores the tree looking for a path to an await node. From this await node, the tree is explored further to find paths to other await nodes, and cycles that contain the await node. If the tree built in this way satisfies the 5 properties of a schedule, then a schedule is built by creating the cycles in our subtree. Otherwise, the algorithm declares the PN as non-schedulable and terminates.

If a finite schedule can be found for a given PN, its properties ensure:

- *Cyclic behavior*: Is guaranteed by properties (4) and (5), because an await state can always be reached from the current state, and an await state is always on a cycle.

- *Correct behavior regardless of the values of data*: Data-dependent constructs are not evaluated during scheduling, but at run-time. Therefore, the schedule describes the behavior of the system for any run-time values of data.

- *Finite memory for communication channels*: The schedule specifies a finite set of reachable markings. Each marking describes the number of tokens for each place, and hence the capacity of all the communication channels for this state. By traversing the finite set of reachable markings, we can compute the upper bound on the number of elements in each communication channel.

3.5 Code generation

The last step of the QSS algorithm is the generation of an efficient implementation of the schedule obtained in the previous phase. A direct translation of the schedule into C code would be possible, but the size of the code would be probably bigger. We will take advantage of the two facts. First, we know that possibly some code fragments will be replicated in different parts of the schedule; we can reduce the size of the generated code by generating a single copy of these code fragments. Second, we have established bounds for all communication channels inside the system. If the source and target of one channel have been merged in a sequential code, we can replace the communication channel by a circular buffer of known size, replacing read/write operations by read/write from the buffer. Moreover, if the buffer has size one, we can replace the communication channel by a single variable, getting a huge speed-up in run-time and allowing compile-time optimizations of the code.

More details on the code generation step can be found in [7].

4 Experimental results

We will illustrate the effectivity of this technique by presenting the results obtained in an industrial example. The example is an MPEG-2 video decoder (Figure 5), developed by Philips and described in [12].

The focus of our example was a subset of this example, formed by about 3000 lines of code in 5 processes: Tisiq, Tidct, TdecMV, Tpredict and Tadd. The Petri Net generated from the FlowC specification had 115 places, 106

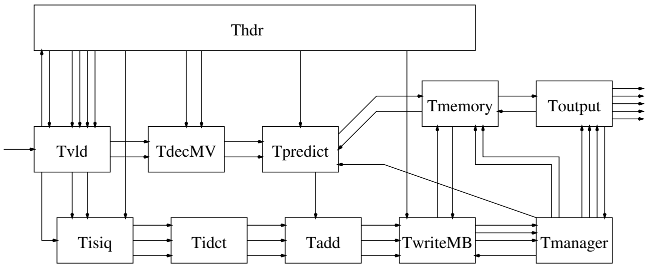


Figure 5: MPEG-2 video decoder block diagram

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

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

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

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

transitions and 309 arcs. QSS generated a single task that replaced the 5 processes in the system.

We compared the performance of the system with the 5 processes scheduled by an operating system versus the task implementation. Processes related with the parsing of the MPEG input stream were left as independent processes to verify the interaction of the generated code with the rest of the system.

Table 1 shows the execution time on a Sun Ultra Enterprise 450 with 512 Mb of main memory for the two implementations. The increase in performance is around 45%. A profile of the execution allowed us to analyze how the execution time was spent, and dividing it in three areas: parsing of the MPEG input, execution of the 5-processes/1-task and operating system intervention. The performance in the last two areas, execution of the 5-processes and operating system intervention, has been enhanced by QSS. Operating system intervention has been greatly reduced because less run-time scheduling is need, and the single task benefits greatly from the elimination of inter-process communication buffers.

Table 2 breaks up further the execution time on the system, analyzes in detail the time spent executing the 5 processes. The execution time is divided in three components: computation, internal communication and external communication. The computation and the external behavior remain the same, as they are not addressed by QSS. However, internal communication has been improved, replacing communication buffers with bounded size 1 by scalar variables. As a last comment, notice that even though it executes faster, the size of the code is bigger in the QSS implementation than in the original one, due to the replication of code in the task.

5 Open problems

The formulation of the QSS problem that has been presented so far is being studied to improve the applicability of the technique. Two of the assumptions that are being made currently are: (a) data information is abstracted from the system and (b) environment works as a black box that produces inputs at regular rates. The abstraction of data implies a loss of information that causes some systems to be considered non-schedulable, and we are working on the incorporation of data-flow analysis into the QSS algorithm to reduce the extent of non-schedulable nets. The black-boxed behavior of the environment makes us conservative when we are building schedules for the system, assuming that inputs are not correlated. This means that some systems could be schedulable in some environments with a given correlation among inputs, e.g. alternated inputs, but are considered *generally* non-schedulable because of being conservative. Our goal will be to capture the behavior of the environment instead of considering it a black box, in order to schedule those systems.

5.1 False path problem

When the QSS algorithm is building the schedules, the data values are abstracted from the Petri Net, and data-dependent choices are abstracted as non deterministic choices. Whenever one of these data-dependent choices is reached, the QSS algorithm proceeds *safely*: it considers that *any* of the paths of the non-deterministic choice could be taken. However, some of these execution paths will not appear at run-time because of the data values of the variables. These execution paths that are considered by QSS (because data information is abstracted) but are unfeasible at run-time are called *false paths*. Figure 6 shows an example of false path in a PN schedule. The false path is a correct sequence of firings of enabled transitions, but transition `tf` will never be fired at run-time because the condition `j>2` will never be true at this point of the execution.

False paths can decrease the quality of the schedule generated for a PN or, even worse, can make the Petri Net non-schedulable. However, modeling data-dependent choices with the information of data values would render the problem undecidable, as it happens with Boolean Dataflow [3].

As an exact analysis is out of the question, we are currently evaluating the incorporation of *approximate data-flow analysis* into the QSS algorithm. Our goal is to work with a *safe approximation* of the set of values at run-time, so that (a) a path can only be discarded if it is a false path although (b) some false paths might not be discarded because our approximation is not precise enough.

Abstract interpretation [10] is an approximate data-flow analysis technique, used widely for program static analysis and program verification. The

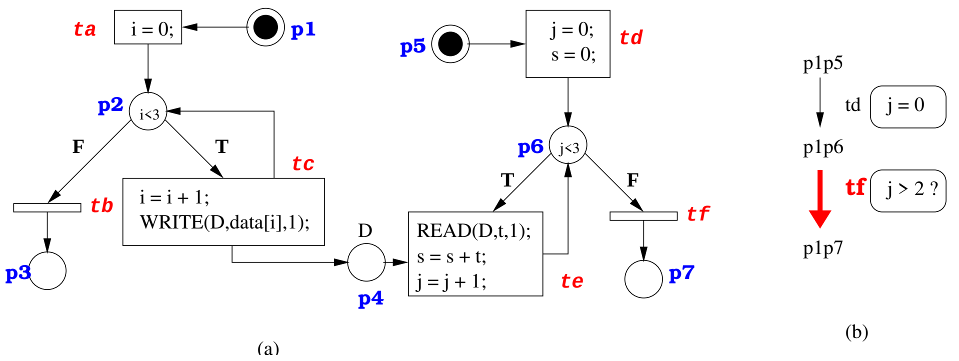


Figure 6: (a) Petri Net to be scheduled, (b) false path in the PN.

uses of abstract interpretation include *range analysis* [10] and *linear inequation analysis* [11]. In many cases, this information is enough to decide the which branch of a data-dependent construct is chosen at run-time. Thus, incorporating abstract interpretation into the algorithm that generates the schedules will let us prune some false paths while the schedules are computed. The extension will yield (a) better schedules for schedulables PNs and (b) schedules for some nets that were previously not schedulable.

5.2 Abstraction of the environment

Currently, the interaction between the system and the environment is performed using *input* and *output* ports of the system. Input ports can be *uncontrollable*, if the environment can send objects to them at any time, or *controllable*, if the environment can decide the moment when the objects are received. Objects in output ports can be read from the environment at any time. The behavior of the environment is assumed to be uncontrollable, i.e. there is no known relation between inputs, rates, etc. The only assumption on the behavior of the environment is that the system is faster that the environment, i.e. the system processes inputs at a faster rate that the rate of production of inputs. Other aspects of the behavior of the system, like relation between inputs, are abstracted from our model and ignored in the QSS algorithm.

However, in some systems we know more information about the behavior of the environment. Adding this information into the model might be valuable for the QSS algorithm. Consider, for example, the system in figure 7. The system on its own is not schedulable, as there is no finite bound on the capacity of the places `data` or `in` if the environment can fire them in any order and rate. However, consider that we model the behavior of the environment as a Petri Net that only interacts through input and output places, and that we know that this behavior is the one shown in figure 7. In this case, it is clear that the capacity of places `data` and `in` is bounded and therefore the PN is schedulable.

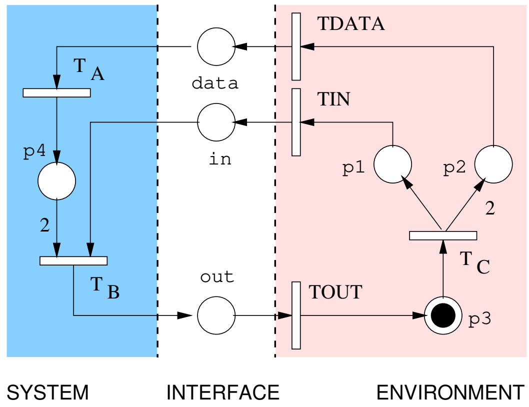


Figure 7: Model that captures system functionality plus environment behavior

Our problem consists on adapting the QSS algorithm to work on a system formed by two components, *system* and *environment*, with the hope that environment information can help the system be schedulable. Both components are modeled by a PN, that describes the functionality of the system and the (possibly abstracted) behavior of the environment, respectively. The QSS algorithm will have to synthesize software that implements the functionality of the system, i.e. no code will be generated for the environment.

The interaction between the two components is limited to adding/removing tokens from input and output places:

- System and environment share all input and output places; they do not share any other place or transition.
- System (environment) can only consume (produce) tokens from (to) input places, and conversely for output places.
- Variables in the system cannot be modified by firing transitions in the environment, and viceversa.

The consequence of this rigid interaction is that firing a transition in the environment does not have any effect on the system, other than possibly enabling a transition that depends on the input places. The same situation happens from the point of view of the environment: firing a transition in the system has no effect on the environment other than enabling a transition that depends on the output places.

This means, that, from the point of view of the system, all transitions fired in the environment are *silent*, except for those that add tokens to input places, the *input transitions*. As we will synthesize code only for the system and not for the environment, we can abstract from the traces of execution the silent transitions of the environment. The transitions that remain in the trace will be the inputs and the transitions of the system that react to this input. Therefore, the traces of execution can be considered as a sequence of *input/response*. This sequence models the correlation among the inputs from the environment. Figure 8 shows an example of such an abstraction as a sequence of input/response; this sequence reflects the dependency among inputs `data` and `in` modeled in figure 7.

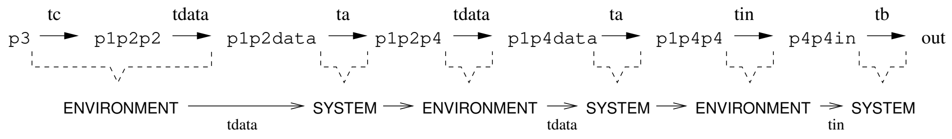


Figure 8: Interaction between the system and the environment described in Figure 7

If the QSS algorithm builds schedules as this sequences input/response, the correlation among inputs will be reflected in the schedules and therefore some systems that were not schedulable might be so inside particular environments.

Another very important benefit that can be obtained from this model is that it allows partitioning of a system. Currently, when we want to synthesize code for a big system, we must do so for the whole system. Modeling an environment allows us to *partition* the system in several subsystems, where *the environment of a system consists of the rest of subsystems.* By applying Petri Net reductions [2] and abstracting irrelevant transitions and places, we can compute an abstracted environment for each subsystem, using it to perform QSS on the subsystem. In this way, we can synthesize code for one subsystem at a time, instead of having to synthesize code for the whole system. The appproach of dividing a Petri Net into two components (system and environment), was already used in [9] for the verification of properties on the execution of the system PN.

6 Conclusion

We have presented a method for the generation of a software component of an embedded system. The method generates a set of tasks, that are invoked when an input event is received. These tasks process the input ensuring that buffer overflow is not possible. The benefits from this approach versus using run-time scheduling are the reduction of memory requirements for communication buffers, the reduction of operating system overhead due to context switching and the generation of code which is more prone to compile-time optimization. Results from an industrial example show an increase in performance of about 45%.

The method uses Petri Nets as the underlying model. Data-dependent choices are modeled by labeling places with boolean expressions, and data manipulation is modeled by labeling transitions with statements of a C language augmented with communication constructs (*FlowC*).

There are still some open problems, because the QSS algorithm fails to schedule some systems. As the problem has been proved undecidable, the goal is to find an approach that let us schedule a relevant subset of those non-schedulable systems. Two promising approaches involve using approximate data-flow information in the construction of the schedules, and building and analyzing a model of the environment. The latter approach is also interesting because it provides a way to synthesize code for a subset of a system, by partitioning a system in two components: the "system", i.e. the parts of the system that we are synthesizing, and the "environment", i.e. the rest of the system.

References

[1] A. Aho, R. Sethi and J. D. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, Reading, MA, U.S.A, 1986.

[2] T. Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, vol 77, NO. 4, 1989.

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

[4] F. Ballarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. M. Sentovich and K. Suzuki. Synthesis of Software Programs for Embedded Control Applications. In *IEEE Transaction on Computer-Aided Design of Integrated Circuits ans SYstems*, vol 18, NO. 4, 1999.

[5] S. Edwards, L. Lavagno, E. A. Lee, A. Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. In *Proceedings of the IEEE*, vol 85, NO. 3, 1997.

[6] M. Sgroi, L. Lavagno and A. Sangiovanni-Vincentelli. Formal Models for Embeddes System Design. In *IEEE Design&Test of Computers*, vol 17, NO. 2, 2000.

[7] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe and A. Sangiovanni-Vincentelli. Task Generation and Compile-Time Scheduling for Mixed Data-Control Embedded Software In *Proceedings*

of the 37th Design Automation Conference, pages 489–494, Los Angeles, California, 2000.

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

[9] A. Valmari. Compositional Analysis with Place-Bordered Subnets. In *15th International Conference on Application and Theory of Petri Nets*, Zaragoza, Spain, 1994.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY, USA.

[11] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY, U.S.A.

[12] P. van der Wolf, P. Lieverse, M. Goeal, D.L. Hei and K. Vissers. An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign*, pages 33–37, May 1999.

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

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

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