

Scheduling Acyclic Branching Programs on Parallel Machines

Marius Bozga, Abdelkarim Aziz Kerbaa, Oded Maler

▶ **To cite this version:**

Marius Bozga, Abdelkarim Aziz Kerbaa, Oded Maler. Scheduling Acyclic Branching Programs on Parallel Machines. 25th IEEE Real-Time Systems Symposium RTSS 2004, Dec 2004, Lisbon, Portugal. pp.208-217. hal-00374877

HAL Id: hal-00374877
https://hal.science/hal-00374877

Submitted on 10 Apr 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Scheduling Acyclic Branching Programs on Parallel Machines*

Marius Bozga, Abdelkarim Kerbaa and Oded Maler
VERIMAG
2 av. de Vignates, 38610 Gières, France.
{**bozga,kerbaa,maler**}**@imag.fr**

Abstract

In this paper we address the following problem: given an acyclic program scheme with if-then-else *control structures, together with the duration of each procedure, and given an architecture consisting of n identical processors, compute offline a scheduling policy that will guarantee minimal execution time (in the worst-case) for the entire program on this architecture. Since this is a problem of scheduling under uncertainty (the results of the branching decisions are not known in advance) it cannot be solved in a satisfactory manner using static or fixed priority schedulers but rather requires a state-dependent* scheduling strategy. *We use timed automata technology to derive such strategies using algorithms for finding shortest paths on game graphs.*

1. Motivation

The class of real-time applications that motivates this work consists of systems that interact with a physical environment and that have to perform computation based on sensory inputs that arrive on a periodic basis. Typical application domains include signal processing, such as radar, and realization of control loops in avionic and automotive domains. The philosophy underlying the development of such systems is a bit different from that used in the "operating systems" genre of real-time systems research.

To begin with, and in contrast with other approaches where the computations to be performed are modeled as *independent* tasks, whose only interaction is via conflicting demands for resources, we view tasks as admitting interrelationships such as precedence and conditional dependency. For example, various control tasks may need to wait for the results of some filtering algorithm before being executed and, moreover, different outcomes of the algorithm may invoke different follow-up computations. Secondly, unlike interactive systems, the major occupation of the class of systems we consider is with periodic, rather than aperiodic, computations. Finally such systems typically avoid the scheduling services of real-time operating systems, albeit their potential performance amelioration. This is partly due to historic reasons and partly because in safety-critical applications, being predictable with respect to timing is considered important while dynamic scheduling policies are perceived as risky and hard to validate. The main goal of this paper is to introduce a scheduling methodology which is appropriate for this type of applications (other efforts in this direction have been made within the *time-triggered architecture* [KB03]).

To explain better our point of departure, let us focus on the development process of control programs written using the synchronous data-flow language Lustre, a special-purpose language tailored for control applications [HCRP91]. Lustre underlies the commercial programming environment SCADE, used for writing parts of the control loops of the Airbus. The philosophy of Lustre is quite simple. Instead of programming by hand the various tasks that realize the control loops, and delegating the handling of their invocation and inter-dependencies to a real-time operating system, the whole application is written in a uniform and modular data-flow style which resembles the block diagrams used by control engineers. In fact, Lustre is very similar to the discrete-time subset of Simulink and an automatic translator from Simulink to C via Lustre has been introduced recently [CCM$^+$03]. During compilation this high-level description is transformed into a single C program that runs on the target machine, with a minimal intervention of an operating system (only I/O and real-time clock services are needed). The advantage of this approach is summarized by the the slogan "what you compile is what you execute": the whole operational semantics resides inside the program itself rather than

* This work was partially supported by the European Community projects IST-2001-35304 AMETIST (Advanced Methods for Timed Systems), IST-2001-38117 RISE (Reliable Innovative Software for Embedded Systems) and IST-2001-33520 CC (Control and Computation) and by a grant from Intel.

being shared with some third-party software with an obscure semantics.

The Lustre compiler generates a single loop C code without too much effort to optimize execution time using scheduling. While this is not such a big issue in critical systems such as in avionics, where the the total computation time of the control loop does not exceed the platform capabilities, other domains which are more computationally expensive (signal processing) or that use cheaper execution platforms (automotive, consumer electronics) require more aggressive scheduling without compromising semantic clarity.

As a starting point we consider an acyclic program which has to be invoked periodically and perform some set of inter-related tasks. We want to minimize the execution time of this program assuming an execution platform with a given number of identical processors. Finding this minimum allows us to determine the highest frequency in which the program can be executed. In real-time systems the problem is sometimes posed in a slightly different form where the frequency is specified (typically using "deadline" and "release time" constraints) and a schedule is sought that satisfies these constraints. Since in our search-based approach there is not much difference between optimization and constraint satisfaction problems, we prefer to concentrate on the former which is more generic. As the reader will see, adding constraints to our approach will only reduce the degrees of freedom of the scheduler and will facilitate the search for the optimal scheduling strategy.[1]

We assume that the (worst-case) execution times of all atomic tasks are known and that the only uncertainty in the system is due to the fact that *not all* tasks need to be executed at every invocation (instance) of the program. Whether or not some task should be executed is not known in advance but is revealed as the computation goes on. Hence the program admits a finite number of execution "scenarios", each corresponding to a subset of the set of tasks.

A naive approach to solve this problem would be to enumerate all instances and solve a (deterministic) scheduling problem for each of them, but this approach ignores the dynamic nature of the uncertainty and assumes a "clairvoyant" scheduler who sees into the future. The optimal schedule achieved by such a scheduler gives only a *lower-bound* on the worst-case computation time of the entire program. On the other hand, the fact that the uncertainty is *bounded* allows us to compute a dynamic scheduling strategy *offline*, without the overhead associated with online re-scheduling. The scheduler can then be implemented as a simple add-on to the compiled code which is invoked when tasks terminate and decides, based on a pre-computed look-up table, which tasks to execute next (or to wait for the next event). In the next section we give a more detailed yet intuitive explanation of the abstract problem that we solve.

2. Introduction

Consider the following program scheme:

prog1: input y
$x_0 := f_0(y)$
$x_1 := f_1(y)$
if $x_0 = 0$
then
  $x_2 := f_2(y)$
  $x_3 := f_3(y)$
  $x_4 := f_4(x_2, x_3)$

Each f_i is a function (task) with a known computation time. All functions except f_4 depend only on some inputs available when the program is invoked and hence can be executed immediately. The functions have no side effects and their only dependencies are via argument passing. Using data-flow analysis we can deduce that f_4 cannot be executed before both f_2 and f_3 terminate. In addition, these three statements are executed *conditionally*, only if $x_0 = 0$ holds, a fact to be revealed only after the termination of f_0. The whole situation can be captured by the conditional precedence graph of Figure 1 where each statement $x_i := f_i$ is represented as a task p_i with a given duration and the condition $x_0 = 0$ is modeled as a special Boolean task b with a zero duration. The precedence constraints between tasks are drawn as arrows and the conditional invocation of p_2, p_3 and p_4 is represented by their inclusion in the left side of the *to* b *or not* b box.[2]

Suppose we want to execute this program as quickly as possible on an architecture with two processors, assuming that communication is for free.[3] At time $t = 0$ we have two tasks, p_0 and p_1, ready and we can start executing both. If after the termination of p_0 at $t = 3$, b evaluates to false, then all we need to do is to wait for the termination of p_1 at $t = 7$. If, however, b evaluates to true, p_2 and p_3 become enabled, but since p_1 still occupies one processor (we

1 The problem of scheduling a Lustre program realizing a multi-rate control system with tasks that need to be executed periodically with different periods can be transformed into a similar acyclic scheduling problem using the following standard transformation. Suppose we have k tasks, $p_1, \ldots p_k$ with (normalized) respective periods $\pi_1, \ldots, \pi_k$. Let π be the least common multiplier of $\pi_1, \ldots, \pi_k$. We then construct a "main loop" of the program with π/π_i copies for each task p_i, with some additional constraints on the duplicate tasks so that they are forced to execute in their corresponding sub-cycles.

2 Unfortunately not all conditional precedence graphs can be drawn so neatly.

3 This assumption can be relaxed at the price of complicating the presentation as well as the computation (because some of the symmetry in the system is lost).

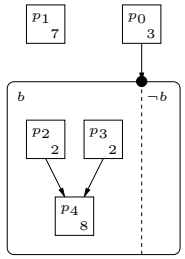


Figure 1. A conditional task graph representation of the program.

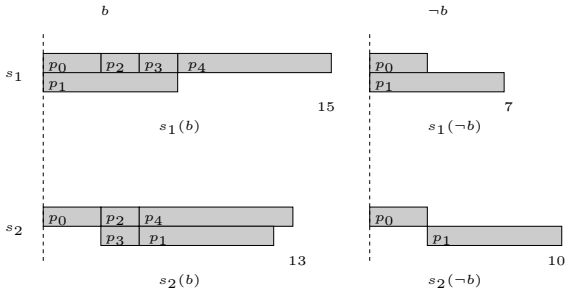


Figure 2. The results of applying strategies s_1 and s_2 to instances b and $\neg b$.

assume no preemption), we can only execute them sequentially, a fact that will delay the execution of p_4 resulting in termination at $t = 15$. The only reasonable alternative to this strategy is to postpone the execution of p_1 until $t = 3$ and then base our decision on the evaluation of b. If it turns out to be false, we start p_1 and terminate with a slight delay at $t = 10$. If, however, b is true, we have two free processors on which we can execute p_2 and p_3 in parallel and only then execute p_1 and p_4 and terminate at $t = 13$. The schedules obtained by these two strategies (we call them s_1 and s_2, respectively) on the two cases (b and $\neg b$) are illustrated in Figure 2.

Which strategy do we prefer? The answer depends on our evaluation criteria. If we want to be optimal with respect to the *worst case*, we will prefer strategy s_2 because $\max\{10, 13\} < \max\{7, 15\}$. If, however, we estimate the probability of b to be true by $\lambda \in [0, 1]$ and want to optimize with respect to the *average case*, we should compare the expected termination times of the two strategies, that is, $(1 - \lambda) \cdot 10 + \lambda \cdot 13$ and $(1 - \lambda) \cdot 7 + \lambda \cdot 15$, in order to choose. It is not hard to see that s_1 is preferable when $\lambda < 3/5$ and that s_2 is preferable otherwise.

In this paper we build a framework in which problems of this kind can be formulated and solved. It consists of the following ingredients:

1. A formal model, conditional precedence graphs, for describing sets of tasks related by precedence constraints and conditional execution.
2. A translation between such models into timed game automata such that all feasible schedules correspond to runs of the automaton. Optimal scheduling is then phrased as finding shortest paths in these automata.
3. Reduction of this problem to finding shortest paths in discrete weighted game graphs.
4. Development of a heuristic forward search algorithm that can find nearly-optimal solutions for that problem.

A prototype implementation of this framework has been built and experiments with randomly-generated instances of the problem have been performed.

3. Conditional Precedence Graphs

The task graph (see, [KI99]) is a commonly used model for describing precedence constraints between tasks. It is essentially a triple $G = (P, \prec, d)$ where P is a set of tasks, $\prec$ is a strict partial-order relation on P with the intended meaning that $p \prec p'$ if p' cannot be executed before the termination of p, and $d : P \to \mathbb{N}$ is a duration function specifying the execution time of each task. The task graph scheduling problem is concerned with finding an optimal schedule for such tasks on a given number of identical machines. In [AKM03] we have shown how to translate this problem into finding shortest paths in timed automata, and were able to find optimal and nearly-optimal schedules for graphs with several thousands of tasks. Since our approach is based on search, it was also easy to incorporate additional features such as deadline and release time constraints into our tool. In this section we extend this model to express conditional execution. This is done by introducing a special type of Boolean tasks with the following features: 1) They can be preceded by other tasks; 2) They take zero time to execute;[4] 3) They terminate with a result which is either true or false; 4) The execution of other tasks may depend on the results of the Boolean tasks;

To express the activation conditions of tasks we will use functions over a set B of Boolean variables that encode the results of the Boolean tasks. To simplify the presentation we restrict ourselves to the class $\mathcal{F}(B)$ of functions that can be written as a conjunction of positive and negative occurrences (literals) of distinct Boolean variables, e.g.

4 This can be relaxed if testing the condition takes non-negligible amount of time, but such tests can be decomposed into an ordinary task and a zero duration test.

$b_1 \wedge \neg b_2 \wedge b_3$. We denote by $V(f)$ the set of variables appearing in f. The partial order on Boolean functions is defined as $f \leq f'$ if for every Boolean vector v, $f(v) \leq f'(v)$. Syntactically this means that the set of literals in f is a superset of those of f'. We say that f and f' are contradictory if $f \wedge f' =$ false, which is the case when at least one variable appears positively in one and negatively in the other. Note the conjunctions can be evaluated to true only if *all* variable values are known, while their falsity can be sometimes deduced from partial information.

Definition 1 (Conditional Precedence Graphs) *A conditional precedence graph (CPG) is* $G = (P, B, \prec, A, d)$ *where* $P = \{p_1, \ldots, p_n\}$ *is a set of tasks,* $B = \{b_1 \ldots, b_m\}$ *is a set of Boolean tasks,* $\prec$ *is a strict partial order precedence relation on* $P \cup B$*,* $A : P \cup B \rightarrow \mathcal{F}(B)$ *is an activation function assigning a Boolean function over* B *to every task, and* $d : P \rightarrow \mathbb{N}$ *specifies task durations.*

We use notation A_p for $A(p)$ and say that task p is *less general* than p' if $A_p < A_{p'}$. We denote this fact by $p \sqsubset p'$. We say that a Boolean task b influences a task p if $b \in V(A_p)$ and denote it by $b \rightharpoonup p$.

Definition 2 (Consistent CPG) *A CPG is* consistent *if the following holds for every* $b \in B$*, and* $p \in P \cup B$*:*

- *No speculation: if* $b \rightharpoonup p$ *then* $b \prec p$*. No task can be executed before it is known whether it has to be executed.*[5]

Remark: We allow consistent CPGs to include precedence $p' \prec p$ when $p' \sqsubset p$, that is, p may depend on a task p' which might not be executed in all situations where p is. We interpret it as a *conditional dependency*, that is p' needs to wait for p only when A_p evaluates to true. Nevertheless, we disallow precedence between tasks whose activation functions are contradictory.

The definition of a feasible schedules for ordinary deterministic task-graph problems is simple. It is an assignment of start times to all tasks such that precedence constraints are satisfied and that the number of tasks active simultaneously at every moment is bounded by the number of machines. The adaptation of the definition to CPGs is more involved because different values of the B variables correspond to different sets of tasks to be executed.

An *instance* of the scheduling problem is an augmented Boolean vector $v : B \rightarrow \{0, 1, \star\}$ where $v(b) = \star$ ("don't care") indicates that $A_b(v)$ is false and b need not be executed. Such situations may occur when the program admits nested *if* statements. A partial instance is an instance which may be undefined for some variables whose values are not known yet. We say that v' *extends* v if it agrees with v on all variables defined in v. The set of tasks associated with an instance v is

$$P_v = \{p \in P \cup B : A_p(v) = true\}.$$

A *schedule* for an instance v is a function $st : P_v \rightarrow \mathbb{R}_+$ indicating the start times of tasks. From st we can derive for each task its termination time, $en(p) = st(p) + d(p)$ and its execution interval $I(p) = [st(p), en(p)]$. The number of active tasks at time t is $\beta(t) = |\{p : t \in I(p)\}|$.

Definition 3 (Feasible Schedules) *A schedule* st *for an instance* v *is feasible on an architecture with* k *machines if*

1. *Precedence: for every* $p \in P_v$*,* $st(p) \geq \max\{en(p') : p' \in P_v \wedge p' \prec p\}$*. A task may start only after all its predecessors have terminated.*
2. *Resource constraints: for every* $t \in \mathbb{R}_+$*,* $\beta(t) \leq k$*. No more than* k *tasks may be active simultaneously.*

The length of a schedule is defined as the termination time of the last task, that is, $\max_{p \in P_v} en(p)$. We would like to obtain schedules that are optimal for all instances, but since instances reveal themselves progressively as more Boolean tasks are evaluated, we should restrict ourselves to *causal scheduling strategies* that can base their decisions only on information available at decision time.

In the last couple of years we have developed a framework for expressing and solving dynamic scheduling problems using timed automata [A02], [AAM03], [AM02] [AKM03]. The timed automaton is the natural tool for modeling the evolution of the *state* of the scheduling problem as a result of discrete actions (starting or ending a task, revealing the value of a Boolean task) and of the passage of time. Before describing how tasks are modeled as timed automata, let us give an informal explanation of the *state-space approach* to modeling scheduling problems. At any given moment the state of a scheduling problem, of the type definable by a CPG, consists of the following information:

1. Which tasks have already been executed, and for the Boolean tasks also what their result was.
2. Which tasks are currently executing and for how long.
3. For which tasks it is known whether they should be executed.
4. Which tasks, among those that should be executed, are enabled for execution (all their predecessors have terminated and not all machines are busy).

5 This assumption can be relaxed if we want to move to the realm of *speculative execution*, used extensively in hardware. The idea is that if you have many processors you may save time by executing alternative conditional branches in parallel and then using only the outcome of the branches that really need to be executed. In that case we replace non-speculation with the weaker *non circularity* condition: if $p \preceq b$ then $b \not\rightharpoonup p$. In other words, the termination of a task cannot be prerequisite for determining whether it is to be executed.

The state of the schedule determines which future evolution is possible. Passage of time increases the elapsed execution time of active tasks and allows them eventually to terminate. Termination of tasks may cause the evaluation of Boolean tasks and make some other tasks enabled. Starting a task, an action performed by the scheduler, moves a task from the waiting list to the active list and resets its timer. These actions belong to three categories:

1. *Deterministic actions*: these are actions that will always happen at certain states. They include termination of a task p exactly $d(p)$ time after its initiation, and the re-evaluation of a task activation function when some new Booleans terminate.
2. *Scheduler actions*: these are the decisions of whether or not to start an enabled task.
3. *Adversary actions*: the choices of the results of Boolean tasks on which we have no control. Note that they are, nevertheless, deterministic with respect to time and happen immediately after they become enabled.

A scheduling strategy is thus a function that assigns to each state of the schedule one of the scheduler actions enabled at this state, including the special waiting "action" which means to do nothing and wait for the next event, while the active tasks keep on executing.

4. Modeling with Timed Automata

Timed automata [AD94] are automata augmented with continuous clock variables which operate in the dense time domain. A behavior (run, execution) of the automaton consists of an alternation of *time passage* periods where the automaton stays in the same state and the clock values grow uniformly, and of *instantaneous transitions* that can be taken when clock values satisfy certain conditions and which may reset some clocks to zero. In the context of scheduling, a clock reset when a certain task was started records at any moment the time spent so far in execution, and its value determines the ability of the task to take the transition that corresponds to termination.

We model scheduling problems defined by CPGs as a product of interacting timed automata. For each ordinary task p we construct a timed automaton $\mathcal{A}_p$ with four states and one clock c as depicted in Figure 3-(a). State $p?$ is the initial state where it is not known yet whether p is to be executed. Once A_p evaluates to true, the automaton moves to a waiting state $\overline{p}$. We *can* leave this state and move to the active state p as soon as the condition Π_p holds, where Π_p is a conjunction of conditions indicating that for every $p' \prec p$, automaton $\mathcal{A}_{p'}$ is in its terminal state. Whether or not to take this transition when Π_p holds is a *decision of the scheduler* and when it is taken the clock is reset to zero. After spending $d(p)$ time in the active state the automaton moves to the terminal state $\underline{p}$. If A_p evaluates to false, the automaton goes from $p?$ directly to $\underline{p}$.

For each Boolean task b we build an automaton (with no clock) $\mathcal{A}_b$ as in Figure 3-(b). This automaton has three terminal states, the state $b\star$ which indicates that the activation condition of b is false and hence it is not executed, and states b and $\neg b$ to which the automaton may move when its activation condition is true and all its predecessors have terminated. As mentioned earlier, the choice between these two transition is the source of uncertainty in the scheduling problem.[6] The evaluation of activation conditions of other tasks that mention b is done on the basis of the states of the corresponding $\mathcal{A}_b$ automata, where $b?$ is interpreted as "unknown" and $b\star$ as "don't care".

The product of all these automata (excluding states where more than k automata are in their active state) is a timed automaton in which every global state is a tuple consisting of the state of each task, including a clock for each task which is in its active state. The runs of this global automaton correspond to all the feasible schedules under all possible scheduling strategies and all adversarial choices. Readers interested in more formal details may look at [AAM03].

To get the intuition let us look at Figure 4 which shows a fragment of the global automaton obtained from the CPG of Figure 1. The difference between strategies s_1 and s_2 is manifested in state $(p_0, \overline{p}_1)$ where the scheduler needs to choose between starting p_1 (left sub-automaton) or waiting for the termination of p_1 (right sub-automaton). The termination of p_0, which leads to the choice of the value of b (dashed arrows), happens at different states in each case. For s_1, this happens when p_1 occupies one machines and consequently, when b is true p_2 and p_3 can only be executed sequentially, leading to a run of length 15. For s_2 this happens at state $\overline{p}_1$ and both p_2 and p_3 can be executed in parallel[7] leading to a schedule of length 13.

Timed automata may have an uncountable number of executions. For example, the $start_1$ transition from state $(p_0, \overline{p}_1)$ can be taken at any time before condition $c_0 = 3$ becomes true, i.e. anywhere in the interval $[0, 3]$. A key result of [AAM03] allows us to consider only a *finite subset* of the runs that we call *non-lazy runs*. A lazy run is a run in which at some state the scheduler hesitates some time before starting a task, while the global state remains the same during that waiting period. Such a laziness could be use-

6 Note also that these two branches never meet again, a fact the induces a special structure of the product automaton which distinguishes it from other types of game graphs.

7 The fact that the $start_2$ and $start_3$ are executed one "after" the other is just an artifact of the interleaving semantics; No time passes between these two transitions and they happen at the same metric time.

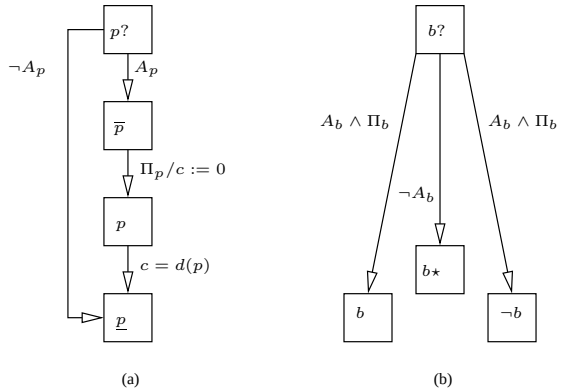


Figure 3. Modeling tasks with timed automata: a) ordinary tasks; b) Boolean tasks.

ful if something has changed during waiting, for example a new task became enabled or a value of a Boolean task was revealed. But if the delayed action is taken in the *same global state*, the waiting is useless and the schedule can be replaced by another schedule of a lesser or equal length in which the lazy action is taken as soon as it is enabled.

Restricting the search to non-lazy runs amounts to transforming the timed automaton into an ordinary finite-state automaton with numerical weights associated with its transitions, and facilitates the application of standard shortest path algorithms. Whenever a global state has several outgoing transitions, the continuations to consider are those in which a $start$ transition is taken immediately, and those in which time (and clock values) advances by the exact amount needed to satisfy the condition for the nearest end transition. State $(p_0, \overline{p}_1)$, for example, has two continuation, one is a result of starting p_1 immediately and the other is the result of waiting 3 time units until $c_0 = 3$ and the end_0 transition is taken.

5. Shortest Paths in Game Graphs

5.1. Game Graphs and Exhaustive Search

Due to the difference in the "ownership" of the $start$ transitions (scheduler) and the transitions of Boolean tasks (environment), the object obtained after restricting the timed automaton to non-lazy runs is a kind of a game graph, also known as AND-OR graph [N71], [Z99], or alternating automaton. Such a graph has two types of nodes, OR nodes where the scheduler ("player 1") chooses an enabled transition, and AND nodes where the adversary chooses between the b and $\neg b$ transitions. Figure 5 shows part of the game graph for our example. The outcome of applying a particular strategy is represented as a sub-graph

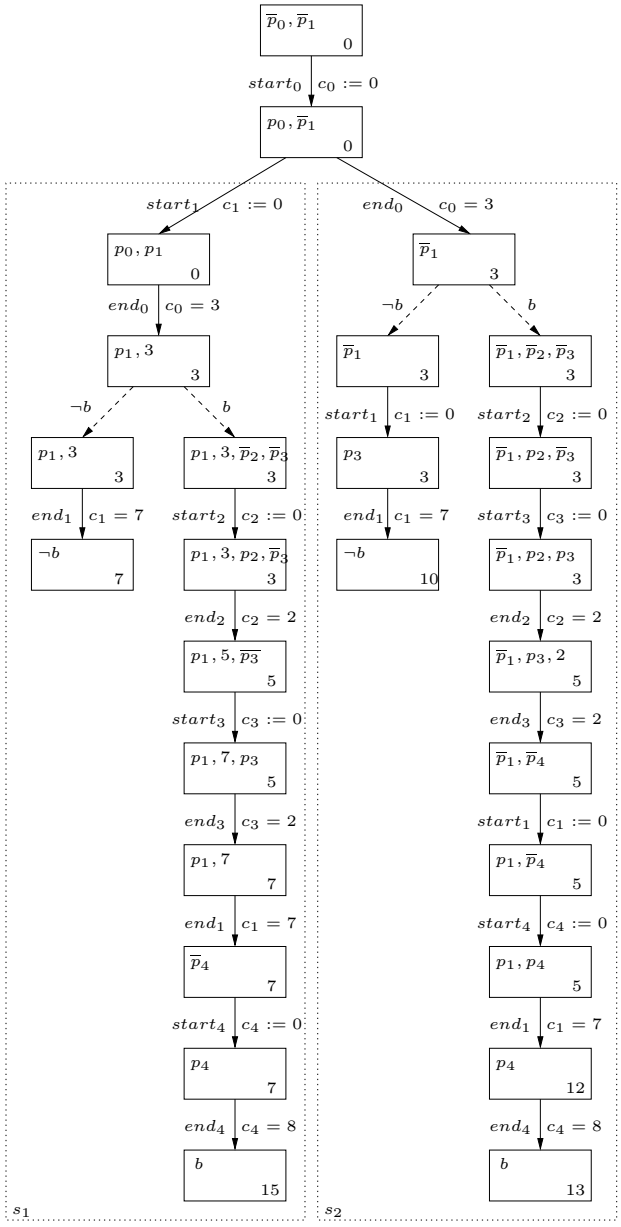


Figure 4. Part of the global automaton for the example of Figure 1. In the states we write only the waiting tasks, the executing tasks and the values of their non zero clocks. The numbers on the lower right corners stand for the total elapsed time to reach the state via a non-lazy run. The branches correspond to the schedules of Figure 2.

rooted at the initial state in which every OR state has exactly *one* successor and every AND node has *all* its successors (see Figure 5). The worst case performance of such a strategy, also known as its *value*, is the length of the longest path in that sub-graph.

Since there are finitely many strategies, one could enumerate and evaluate all of them, but a more intelligent way to find the optimum is to exploit the structure of the space of strategies. Taking the left branch at q_1 is the beginning of the exploration of all scheduling strategies that decide to wait in this state. States q_5 and q_6 are two possible outcomes of this partial strategy, and any strategy that extends it has to choose what to do in each of them. Taking the right branch at q_1 starts the exploration of the rest of the strategies, each of which has to be defined for q_2, and those that take there the right branch need to be defined for q_7 and q_8 and so on. Note that a strategy should be defined only for states reachable while applying that strategy.

Definition 4 (Game Automaton)
A game automaton over a set B of Boolean variables is $\mathcal{M} = (Q, q_0, \Sigma, \delta_\vee, \gamma, \bar{B}, \delta_\wedge)$ where $Q = Q_\vee \cup Q_\wedge$ is a set of states partitioned into OR and AND states, $q_0 \in Q_\vee$ is the initial state, Σ is a set of actions enabled at certain OR states, $\delta_\vee : Q_\vee \times \Sigma \to Q$ is a partial transition function on OR states, $\gamma : Q_\vee \times \Sigma \to \mathbb{R}_+$ is a cost function defined for every q and σ such that $\delta_\vee(q, \sigma)$ is defined. The set $\bar{B}$ consists of positive and negative literals over the B variables and $\delta_\wedge : Q_\wedge \times \bar{B} \to Q$ is the adversary (partial) transition function, defined for each $q \in Q_\wedge$ for exactly one pair of opposing literals.

We assume that the transition graph of the automaton is acyclic and use $\mathcal{M}_q$ to denote the sub-automaton rooted at state q, which represents the "residual" game which remains to be played after reaching q. A state $q \in Q_\vee$ is said to be terminal if $\delta_\vee(q, \sigma)$ is not defined for any σ. A strategy is a partial function $s : Q_\vee \to \Sigma$, represented as a sub-graph in which for every $q \in Q_\vee$ all (q, σ) transitions are removed except for $\sigma = s(q)$. A particular property of the game graphs obtained for CPGs is that all strategies induce sub-graphs which are trees with the same type of branching.

The computation of the optimal strategy is a by product of computing the *value function* $h : Q \to \mathbb{R}_+$ where $h(q)$ is the value of the best strategy for the residual game $\mathcal{M}_q$. This function is defined recursively as:

$$h(q) = \begin{cases} 0 & \text{if } q \text{ is terminal} \\ \min_{\sigma \in \Sigma} \gamma(q, \sigma) + h(\delta_\vee(q, \sigma)) & \text{if } q \in Q_\vee \\ \min\{h(\delta_\wedge(q, b), \delta_\wedge(q, \neg b)\} & \text{if } q \in Q_\wedge \end{cases}$$

There are various ways to compute h, one of them is the *backward value iteration* procedure, also known as *dynamic programming*, which starts with the final states and propagates values according to the definition of h until $h(q_0)$ is defined. A forward depth-first algorithm is obtained by invoking $h(q_0)$ and following literally the recursive definition. If the game graph has a tree structure, this procedure has the same complexity as dynamic programming, however on non-tree graphs, the algorithm can easily become exponential since the same node can be reached via many paths. Consequently we use an algorithm that combines depth-first search with memorization: whenever $h(q)$ is computed for the first time, the result is stored as $E(q)$ and subsequent invocations of $h(q)$ are answered using this value.

Algorithm 1 (Forward Value Iteration)

integer function $h(q)$
if q *is terminal* **return**(0)
elsif $E(q)$ *is defined* **return**$(E(q))$
elsif $q \in Q_\vee$ **then**
begin
  $E{:=}\infty$
  for *every* σ *such that* $\delta_\vee(q, \sigma)$ *is defined* **do**
    $E' := \gamma(q, \sigma) + h(\delta_\vee(q, \sigma))$
    $E := \min\{E, E'\}$
  $E(q) := E$
  return*(E)*
end
elsif $q \in Q_\wedge$ **then**
begin
  $E{:=}h(\delta_\wedge(q, \neg b))$
  $E'{:=}h(\delta_\wedge(q, b))$
    $E := \max\{E, E'\}$
  $E(q) := E$
  return(E)
end

The derivation of a strategy from the value function is standard: let $s(q) = \sigma$ for every $q \in Q_\vee$ from which the minimum is obtained via the transition σ.

Algorithm 1 is (time and space) linear in the size of the game graph but this is not of much help because the game graph by itself is exponential in the size of the CPG. The largest problems that we could solve with this algorithm had up to 14 tasks and 4 Booleans. We have preferred this algorithm over backward dynamic programming because it is more easily amenable to techniques that find optimal or nearly-optimal solutions without exploring the whole graph.

5.2. Best First Search

Techniques for pruning the search space are based on two related ideas: 1) Do not explore paths that can easily be shown not to lead to an improvement of the value function computed so far; 2) Replace the arbitrary depth-first (or breadth-first) order of exploration by a more "intelligent" policy, which explores the more promising successors

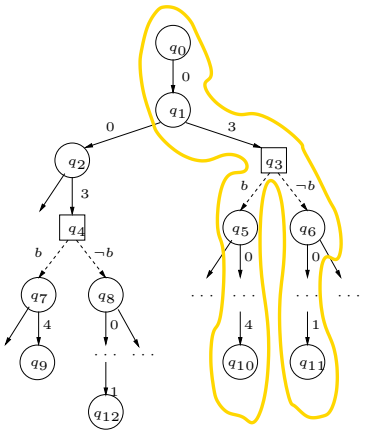


Figure 5. Part of the game graph for our example, with AND nodes denoted by squares. The sub-graph of strategy s_2 is marked.

first. The first part of the methodology is based on an auxiliary *estimation function* $\bar{\mu}(q)$ which approximates $h(q)$. If this function is *optimistic*, that is, $\bar{\mu}(q) \leq h(q)$ for every q, we need not explore σ-successors of an OR node q satisfying $E \leq \gamma(q,\sigma) + \bar{\mu}(\delta_\vee(q,\sigma))$ where E is a value obtained via an already explored successor. We refer to the above condition as a *safe cutting test*.

5.2.1. Estimation Functions for Conditional Scheduling For unconditional precedence graphs, an estimation function that gives a lower bound on the time remaining until termination from a state, can be constructed by first associating with each task p the length $\mu(p)$ of the longest path in the CPG from p to some terminal task. Then, the estimation $\bar{\mu}$ of the value of the global state, is the maximum of μ over all tasks which are waiting or active in this state.

In the conditional setting this is more involved due to precedence constraints between tasks that have *different activation conditions*. Consequently the distance from a task to termination is not a single number but is *instance dependent*. Let $\mu : P \times \{0,1\}^n \to \mathbb{R}_+$ be a partial function defined over all v such that $A_p(v)$ is true. When $A_p(v)$ is false we use the notation $\mu(p,v) = \perp$. The intended meaning of $\mu(p,v)$ is the total amount of work that needs to be done for instance v before task p has started. Since computation of longest paths[8] is done (explicitly or implicitly) within the $(\max,+)$-algebra we need to extend these two operations to $\mathbb{R}_+ \cup \{\perp\}$ by letting $r + \perp = \perp$ and $\max\{r,\perp\} = r$ for every $r \in \mathbb{R}_+$.

8 The reader should *not* confuse the precedence graph (CPG) on which we compute longest paths, as in PERT, with the transition graph of the game automaton on which we seek shortest paths.

A simple way to understand this function (although not the most efficient way to compute it) is the following: for each instance v let G_v be the sub-graph consisting of the tasks whose activation conditions are satisfied by v. If p does not belong to G_v then $\mu(p,v) = \perp$, otherwise let $\mu(p,v)$ be the longest path in G_v from p to termination. This function can be computed backwards on the whole graph, starting from terminal nodes:

$$\mu(p,v) = \begin{cases} \perp & \text{if } A_p(v) = false \\ d(p) & \text{otherwise} \end{cases}$$

and computing for other nodes as

$$\mu(p,v) = d(p) + \max_{p' : p \prec p'} \mu(p',v).$$

This computation can be done symbolically (and offline) using the syntax of A without necessarily enumerating all instances. From μ we can define an estimation function $\bar{\mu}$ over the states and clock values of the $\mathcal{A}_p$ automaton by letting $\bar{\mu}(p?,c,v) = \bar{\mu}(\overline{p},c,v) = \mu(p,v)$, $\bar{\mu}(p,c,v) = \mu(p,v) - c$ and $\bar{\mu}(\underline{p},c,v) = \mu(p,v) - d(p)$.

The function $\bar{\mu}$ is *optimistic* because it takes into account precedence constraints but ignores resource constraints. In other words it assumes sufficiently many machines so that every task can be executed once all of its predecessors have terminated. A complementary way to obtain lower-bounds on schedule length is to ignore precedence constraints and take into account resource constraints, that is, dividing the total amount of work by the number of machines. The estimation

$$\nu_k = \max_v \sum_{p \in P_v} d(p)/k$$

is equally optimistic as it ignores the possibility that a machine can be idle at certain times because no task is enabled. Like μ, estimation ν can be defined for global states of the game automaton by restricting summation to tasks that have not terminated.

5.2.2. Ordering and Sub-Optimal Solutions As for ordering the successors of an OR node q for the purpose of best-first search, let us first note that they consist of at most k $start$ transitions (where k is the maximal width of the CPG) and one $wait$ transition (letting time pass until the termination of the nearest active task). A natural ordering for the $start$ transitions is to prefer starting p_i over p_j whenever $\mu(p_i,v) > \mu(p_j,v)$, giving priority to tasks that lie along the "critical path" of the problem. Although using best-first ordering turned out to reduce the number of explored states, it did not increase the size of problems that could be solved exactly and we need to resort to heuristic methods that explore small subsets of the search space which are not sufficient to guarantee optimality.

One class of heuristic methods would be the bound a-priori the number of explored nodes and give the best solution found until this number is reached. To avoid getting stuck at a "local" optimum we can modify the algorithm to explore only a fixed-size subset of size w of the successors of each node, to avoid concentration on the "left" part of the search tree. Another type of heuristics is to replace the safe cutting test by a more liberal one and explore nodes only if $\alpha\mu(q) < E$ where α is a number larger than 1 which injects some "realism" into the optimistic estimation function. The results reported in the next section are based on a heuristic that explores the $w = 3$ best (according to μ) successors of each OR node, uses the safe cutting test and, after exploring 80K states, switches to $w = 1$ to complee the exploration of some unfinished bracnches.

6. Experimental Results

We have implemented a tool which reads a CPG, translates it into a product of timed automata and then performs best first search, with and without guarantee for optimality, while generating states on the fly. The estimation function is computed offline and stored as a BDD. We have tested this algorithm on randomly generated CPGs on a 1.7GHz Pentium III machine with 2GB of memory. As mentioned earlier, 14 tasks and 4 Booleans is currently the upper limit for exact methods.

Table 1 gives preliminary results and indications about the applicability of our approach to large problems. For each example we find sub-optimal schedules on architectures with 3, 4 and 5 machines, using heuristic best first search. Since the exact optimum for these problem is too hard to compute, we compare these results with the lower-bound $\max\{\mu, \nu_k\}$. Note that this is not a comparison with the real optimum but with an optimistic estimation of it. We find the results rather encouraging as they demonstrate the ability to compute close to optimal scheduling policies for problems with up to 120 tasks and up to 10 Booleans. We are confident that these results can be still improved significantly by tuning the algorithm, and we look forward for testing it on real, rather than randomly-generated, examples.

7. Discussion

We have presented a comprehensive framework for posing and solving the important problem of optimal scheduling of conditional tasks. The overall structure of the proposed methodology is depicted in Figure 6. Our solution is based on the principle of starting with a rigorous model having a well-defined semantics, and only then moving to heuristic algorithms. Additional features such as release times and deadlines can be added very naturally [AKM03], as well as communication costs. Our methodology can be applied at *any desired granularity*, from simple statements to procedures, and we believe that in the future it will provide a useful tool for compiling embedded software on parallel architectures and perhaps also for processor instruction scheduling. Finding the worst case optimal schedule for a program which is to be invoked periodically can determine the sampling rate (or clock rate, in hardware) with which the program can function.

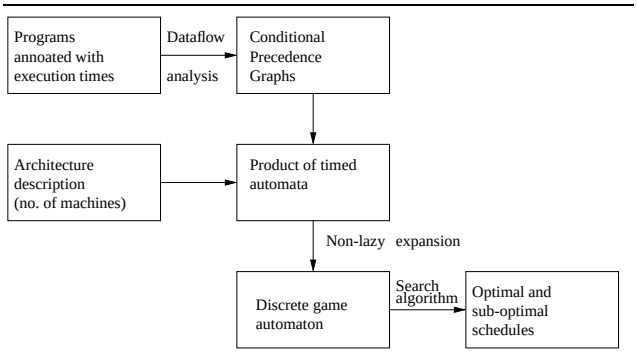


Figure 6. The overall methodology.

The next steps in our work are the amelioration and fine tuning of the search algorithm and the development of a front end for translating programs into CPGs via data flow analysis. Further research directions include the adaptation of the algorithm to average-case analysis, the extension of our framework to speculative executions and the combination of the discrete uncertainty treated in this paper with other features we have dealt with in the past, namely preemption [AM02] and temporal uncertainty concerning the duration of atomic tasks [AAM03].

This work is part of an ongoing effort to enrich scheduling theory with techniques based on timed automata. The reader is referred to [AGP99] and [STY03] for the general approach, to [CPP+01], [KY03] and [AFM+03] for other timed automaton based tools tailored for program scheduling, to [AMPS98] for general controller synthesis for timed automata, to [AM99], [NTY00], and [BFH+01] for algorithms for finding shortest paths and optimal schedules and to [M04] for a unified game theoretic framework for optimal control in the presence of adversaries.

As for other approaches to similar problems we mention the work of [KW02] in the context of hardware/software co-design. Although this work is also geared toward scheduling for conditional execution, it treats a much simpler problem where the only Boolean variables are clocks (in the hardware sense) whose values do not depend on the results of other tasks. Likewise, the approach taken in the time-triggered architecture [KB03] does not consider dependencies among tasks. It determines manually the distribution of tasks on machines and then focuses on finding schedules for the common bus through which they communicate.

k	3				4				5			
(n, m)	len	time	nb st	dev	len	time	nb st	dev	len	time	nb st	dev
$100, 6$	268	2:37	119648	3.07%	198	3:26	126703	1.53%	162	4:10	141319	3.84%
$100, 7$	316	3:36	142602	0.00%	238	4:12	159369	0.42%	192	4:20	158961	1.05%
$120, 7$	357	4:15	143480	0.84%	270	6:20	182788	1.50%	218	3:40	101104	2.34%
$100, 10$	480	6:51	190163	2.78%	371	10:03	244199	6.00%	345	5:58	116089	3.60%

Table 1. Results for applying bounded best first search to some large examples Column `len` indicates the worst-case length of the best solution found, `time` indicates computation time (in minutes), `nb st` is the number of explored states and `dev` is the deviation from the (optimistic) lower-bound.

Acknowledgments Part of the motivation of this work is due to discussions with P. Caspi and S. Tripakis concerning the scheduling of Lustre programs. Helpful comments were given by P. Niebert, A. Asarin, Ph. Gerner, S. Cotton, S. Yovine, A. Curic, R. Rajkumar, I. Broster, A. Pnueli and anonymous referess.

References

[A02] Y. Abdedaïm, *Scheduling with Timed Automata*, PhD Thesis, INPG, Grenoble, 2002.

[AAM03] Y. Abdeddaïm, E. Asarin and O. Maler, Scheduling with Timed Automata, *Theoretical Computer Science* to appear, 2004.

[AKM03] Y. Abdedaïm, A. Kerbaa and O. Maler, Task Graph Scheduling using Timed Automata, *Proc. FMPPTA'03*, 2003.

[AM02] Y. Abdeddaïm and O. Maler, Preemptive Job-Shop Scheduling using Stopwatch Automata, *Proc. TACAS'02*, 113-126, LNCS 2280, Springer, 2002.

[AGP99] K. Altisen, G. Goessler, A. Pnueli, J. Sifakis, S. Tripakis and S. Yovine, A Framework for Scheduler Synthesis, *Proc. RTSS'99*, 154-163, IEEE, 1999.

[AD94] R. Alur and D.L. Dill, A Theory of Timed Automata, *Theoretical Computer Science* 126, 183-235, 1994.

[AFM+03] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, TIMES: a Tool for Schedulability Analysis and Code Generation of Real-Time Systems, *Proc. FORMATS'03*, 2003.

[AM99] E. Asarin and O. Maler, As Soon as Possible: Time Optimal Control for Timed Automata, *Proc. HSCC'99*, 19-30, LNCS 1569, Springer, 1999.

[AMPS98] E. Asarin, O. Maler, A. Pnueli and J. Sifakis, Controller Synthesis for Timed Automata, *Proc. IFAC Symposium on System Structure and Control*, 469-474, Elsevier, 1998.

[BFH+01] G. Behrmann, A. Fehnker T.S. Hune, K.G. Larsen, P. Pettersson and J. Romijn, Efficient Guiding Towards Cost-Optimality in UPPAAL, *Proc. TACAS 2001*, 174-188, LNCS 2031, Springer, 2001.

[CCM+03] P. Caspi, A. Curic, A. Maignan, C. Sofronis and S. Tripakis, Translating Discrete-Time Simulink to Lustre, *Proc. EMSOFT'03*, 84-99, LNCS 2855, Springer, 2003.

[CPP+01] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil and S. Yovine, TAXYS: a Tool for the Development and Verification of Real-Time Embedded Systems, *Proc. CAV'01*, LNCS 2102, Springer, 2001.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, The Synchronous Dataflow Programming Language LUSTRE, *Proc. of the IEEE* 79, 1305-1320, 1991.

[KI99] K.-Y. Kwong, A. Ishfaq, Benchmarking and Comparison of the Task Graph Scheduling, *J. of Parallel and Distributed Computing 59*, 381-422, 1999.

[KW02] A.A. Kountouris, Ch. Wolinski, Efficient Scheduling of Conditional Behaviors for High-Level Synthesis, *ACM Transaction on Design Automation of Electronic Systems* 7, 380-412, 2002.

[KY03] Ch. Kloukinas, and S. Yovine, Synthesis of Safe, QoS Extendible, Application Specific Schedulers for Heterogeneous Real-Time Systems *Proc. ECRTS'03*, 2003.

[KB03] H. Kopetz and G. Bauer, The Time-Triggered Architecture, *Proc. of the IEEE* 91, 112-126, 2003.

[M04] O. Maler, On Optimal and Sub-optimal Control in the Presence of Adversaries, *Proc. WODES'04*, 1-12, 2004.

[N71] N. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.

[NTY00] P. Niebert, S. Tripakis and S. Yovine, Minimum-Time Reachability for Timed Automata, *IEEE Mediteranean Control Conference*, 2000.

[STY03] J. Sifakis, S. Tripakis and S. Yovine, Building Models of Real-time Systems from Application Software, *Proceedings of the IEEE* 91, 100-111, 2003.

[Z99] W. Zhang, *State-Space Search: Algorithms, Complexity, Extensions and Applications*, Springer 1999.