Allocation and Scheduling of Conditional Task Graphs

Michele Lombardi, Michela Milano

DEIS Universita' di Bologna - Viale Risorgimento, 2 40136 Bologna Italy

Abstract

We propose an original, complete and efficient approach to the allocation and scheduling of Conditional Task Graphs (CTGs). In CTGs, nodes represent activities, some of them are branches and are labeled with a condition, arcs rooted in branch nodes are labeled with condition outcomes and a corresponding probability. A task is executed at run time if the condition outcomes that label the arcs in the path to the task hold at schedule execution time; this can be captured off-line by adopting a stochastic model. Tasks need for their execution either unary or cumulative resources and some tasks can be executed on alternative resources. The solution to the problem is a single assignment of a resource and of a start time to each task so that the allocation and schedule is feasible in each scenario and the expected value of a given objective function is optimized. For this problem we need to extend traditional constraint-based scheduling techniques in two directions: (i) compute the probability of sets of scenarios in polynomial time, in order to get the expected value of the objective function; (ii) define conditional constraints that ensure feasibility in all scenarios. We show the application of this framework on problems with objective functions depending either on the allocation of resources to tasks or on the scheduling part. Also, we present the conditional extension to the timetable global constraint. Experimental results show the effectiveness of the approach on a set of benchmarks taken from the field of embedded system design. Comparing our solver with a scenario based solver proposed in the literature, we show the advantages of our approach both in terms of execution time and solution quality.

Key words:
PACS:

1 Introduction

Conditional Task Graphs (CTG) are directed acyclic graphs whose nodes represent activities, linked by arcs representing precedence relations. Some of the activities are branches and are labeled with a condition; at run time, only one of the successors of a branch is chosen for execution, depending on the occurrence of a condition outcome labeling the corresponding arc. The truth or the falsity of those condition outcomes is not known a priori: this sets a challenge for any off-line design approach, which should take into account the presence of such elements of uncertainty. A natural answer to this issue is adopting a stochastic model. Each activity has a release date, a deadline and needs a resource to be executed. The problem is to find a resource assignment and a start time for each task such that the solution is feasible whatever the run time scenario is and such that the expected value of a given objective function is optimized. We take into account different objective functions: those depending on the resource allocation of tasks and those depending on the scheduling side of the problem.

CTG are ubiquitous to a number of real life problems. In compilation of computer programs [13], for example, CTGs are used to explicitly take into account the presence of conditional instructions. Similarly, in the field of system design [39], CTGs are used to describe applications with if-then-else statements; in this case tasks represent processes and arcs are data communications. Once a hardware platform and an application is given, to design a system amounts to allocate platform resources to processes and to compute a schedule; in this context, taking into account branches allows better resource usage, and thus lower costs.

CTG may be used also in the Business Process Management (BPM) [34] and in workflow management [30], as a mean of describing operational business processes with alternative control paths.

For solving the allocation and scheduling problem of CTG we need to extend the traditional constraint based techniques with two ingredients. First, to compute the expected value of the objective function, we need an efficient method for reasoning on task probabilities in polynomial time. For example, we have to compute the probability a certain task executes or not, or, more in general, the probability of a given set of scenarios with uniform features (e.g. the same objective function value). Second, we need to extend traditional constraints to take into account the feasibility in all scenarios.

For this purpose, we define a data structure called Branch/Fork graph - BFG. We show that if the CTG satisfies a property called Control Flow Uniqueness - CFU, the above mentioned probabilities can be computed in polynomial time. CFU is a property that holds in a number of interesting applications, such as for example the compilation of computer programs, embedded system design and in structured business processes.

The paper is organized as follows: Section 2 presents some applications where

CTG is a convenient representation of problem entities and their relations; in Section 3 we provide some preliminary notions on Constraint-Based Scheduling. Section 4 introduces the concept of Conditional Task Graphs, Control Flow Uniqueness, sample space and scenarios and defines the scheduling and allocation problem we consider. In Section 5 we define the data structure used for implementing efficient probabilistic reasoning, namely the Branch/Fork Graph and related algorithms. In Section 6 we use these algorithms for efficiently computing the expected value of three objective function types, while in Section 7 we exploit the BFG for implementing the conditional variant of the timetable global constraint. Section 8 discusses related work and Section 9 shows experimental results and a comparison with a scenario based approach.

2 Applications of CTGs

Conditional Task Graphs can be used as a suitable data structure for representing activities and their temporal relations in many real life applications. In these scenarios, CTG allocation and scheduling becomes a central issue.

In compilation of computer programs [13], for example, CTGs are used to explicitly take into account the presence of conditional instructions. For instance, Figure 1 shows a simple example of pseudo-code and a natural translation into a CTG; here each node corresponds to an instruction and each branch node to an "if" test; branch arcs are label with the outcome they represent. In this case, probabilities of condition outcomes can be derived from code profiling. Clearly, computer programs may contain loops that are not treated in CTGs, but modern compilers adopt the *loop unrolling* [17] technique that can be used here for obtaining cycle free task graphs.

```
if $a = 0$ then
    return error
else
    if $b < 0$ then
        $b = b + 1$
    end if
end if
return $a \cdot b$
```

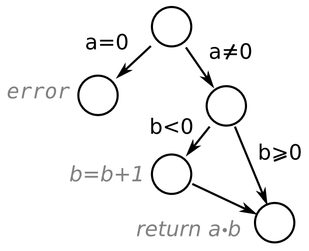


Fig. 1. Some pseudo-code and a its translation into a CTG

Similarly, in the field of embedded system design [39] a common model to describe a parallel application is the task graph. The task graph has a structure similar to a data flow graph, except that the tasks in a task graph represent larger units of functionality. However, a task graph model that has no control dependency information can only capture data dependency in the system specification. Recently, some researchers in the co-synthesis domain have tried to use conditional task graph to capture both data dependencies and control dependencies of the system specification [42], [18]. Once a hardware platform and an application is given, to design a system amounts to allocate platform resources to processes and to compute a schedule; in this context, taking into account branches allows better resource usage, and thus lower costs. However, the presence of probabilities makes the problem extremely complex since the real time and quality of service constraints should be satisfied for any execution scenario. Embedded system design applications will be used in this paper to experimentally evaluate the performance and quality of our approach.

CTG appear also in Business Process Management (BPM) [34] and in workflow management [30] as a mean of describing operational business processes with alternative control paths. Workflows are instances of workflow models, that are representations of real-world business processes [41]. Basically workflow models consist of activities and the ordering amongst them. They can serve different purposes: they can be employed for documentation of business processes or can be used as input to a Workflow Management System that allows their machine-aided execution.

One of the most widely used systems for representing business processes is BPEL [27]. BPEL is a graph-structured language and allows to define a workflow model using nodes and edges. The logic of decisions and branching is expressed through transition conditions and join conditions. Transition conditions and join conditions are both Boolean expressions. As soon as an activity is completed, the transition conditions on their outgoing links are evaluated. The result is set as the *status of the link*, which is true or false. Afterwards, the target of each link is visited. If the status of all incoming links is defined, the join condition of the activity is evaluated. If the join condition evaluates to false, the activity is called *dead* and the status of all its outgoing links is set to false. If the join condition evaluates to true, the activity is executed and the status of each outgoing link is evaluated. CTGs behave exactly in the same fashion and can be used to model BPEL workflow models. In addition CTG provide probabilities on branches. Such numbers, along with task durations and resource consumption and availability can be extracted from process event logs. The CTG allocation and scheduling proposed in this paper can be used in the context of workflow management as a mean to predict the completion time of the running instances, as done in [1], or for scheduling tasks to obtain the minimal expected completion time.

3 Preliminaries on Constraint-Based Scheduling

In this paper we show how to extend constraint-based scheduling techniques for dealing with probabilistic information and with conditional task graphs. We therefore provide some preliminary notions on Constraint-Based Scheduling.

Constraint-Based Scheduling [4] is a subfield of the area of Constraint Programming (CP). Generally speaking, CP is concerned with solving Constraint Satisfaction Problems (CSPs). A CSP is a triple $\langle X, D, C\rangle$, where X is a set of variables, D is the set of their domains and C is a set of constraints; a constraint denotes a relation between the values of the variables it refers to. Solving a CSP consists of assigning values to variables such that all constraints are satisfied simultaneously. In CP, constraints are actively exploited to reduce the domains of the variables and to detect inconsistencies via *constraint propagation*. For example, let $D(X)$ denote tha domain of variable x; then given $D(x) = D(y) = \{0, 1, 2, 3\}$ and a constraint $x < y$, after constraint propagation $D(x)$ is reduced to $\{0, 1, 2\}$ and $D(y)$ to $\{1, 2, 3\}$. Note that detecting all problem inconsistencies is as difficult as solving the original problem. Thus, constraint propagation enforces only partial (namely *local*) consistency [11]; consequently, one needs to perform some kind of search to determine whether the CSP instance at hand has a solution or not.

Scheduling problems over a set of activities are classically modeled in CP by introducing for every activity three variables representing the start time (*start*), end time (*end*) and duration (*dur*). In this context a solution (or schedule) is an assignment of all *start* and *end* variables. “Start”, “end” and “duration” variables must satisfy the constraint $end = start + dur$.

Activities require a certain amount of resources for their execution. We consider in this paper both unary resources and discrete (or cumulative) resources. Unary resources have capacity equal to one and two tasks using the same unary resource cannot overlap in time, while cumulative resources have finite capacity that cannot be exceeded at any point in time.

Scheduling problems often involve precedence relations and alternative resources; precedence relations are modeled by means of constraints between the start and end variables of different activities, while special resource constraints guarantee the capacity of each resource is never exceeded in the schedule; a number of different propagation algorithms for temporal and resource constraints [3,21] enable an effective reduction of the search space. Finally, special scheduling oriented search strategies [4] have been devised to efficiently find consistent schedules or to prove infeasibility.

4 Problem description

The problem we consider is the scheduling of Conditional Task Graphs (CTG) in presence of unary and cumulative alternative resources. In the following, we introduce the definitions needed in the rest of the paper. In Section 4.1 we provide some notions about Conditional Task Graphs, Section 4.2 concerns Control Flow Uniqueness, a CTG property that enables the definition of polynomial time CTG algorithms, Section 4.3 introduces the concept of sample space and scenarios while Section 4.4 describes the scheduling and allocation problem considered in the paper.

4.1 Conditional Task Graph

A CTG is a directed acyclic graph, where nodes are partitioned into branch and fork nodes. Branches in the execution flow are labeled with a condition. Arcs rooted at branch nodes are labeled with condition outcomes, representing what should be true in order to traverse that arc at execution time, and their probability. Intuitively, fork nodes originate parallel activities, while branch nodes have mutually exclusive outgoing arcs.

More formally:

Definition 1 *A CTG is a directed acyclic graph that consists of a tuple* $\langle T, A, C, P\rangle$*, where*

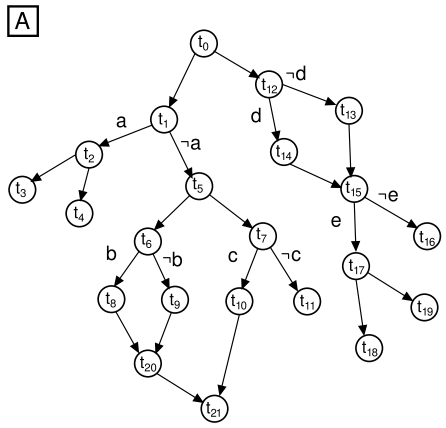
- $T = T_B \cup T_F$ *is a set of nodes;* $t_i \in T_B$ *is called a branch node, while* $t_i \in T_F$ *is a fork node.* T_B *and* T_F *partition set* T*, i.e.,* $T_B \cap T_F = \emptyset$*. Also, if* $T_B = \emptyset$ *the graph is a deterministic task graph.*
- A *is a set of arcs as ordered pairs* $a_h = (t_i, t_j)$*.*
- C *is a set of pairs* $\langle t_i, c_i\rangle$ *for each branch node* $t_i \in T_B$*.* c_i *is the condition labeling the node.*
- P *is a set of triples* $\langle a_h, Out, Prob\rangle$ *each one labeling an arc* $a_h = (t_i, t_j)$ *rooted in a branch node* t_i*.* $Out = Out_{ij}$ *is a possible outcome of condition* c_i *labeling node* t_i*, and* $Prob = p_{ij}$ *is the probability that* Out_{ij} *is true* *(*$p_{ij} \in [0, 1]$*).*

The CTG always contains a single root node (with no incoming arcs) that is connected (either directly or indirectly) to each other node in the graph.

For each branch node $t_i \in T_B$ *with condition* c_i *every outgoing arc* (t_i, t_j) *is labeled with one distinct outcome* Out_{ij} *such that* $\sum_{(t_i,t_j)} p_{ij} = 1$*.*

Intuitively, at run time, only a subgraph of the CTG will *execute*, depending on the branch node condition outcomes. Each time a branch node is executed, its condition is evaluated and only one of its outgoing arcs is evaluated to true. In Figure 2A if condition a is true at run time, then arc (t_1, t_2) status is true and node t_2 executes, while arc (t_1, t_5) status is false and node t_5 does not execute. Without loss of generality, all examples throughout this paper target

A



B

p(a) = 0.5	p(b) = 0.4
p(¬a) = 0.5	p(¬b) = 0.6
p(c) = 0.6	p(d) = 0.3
p(¬c) = 0.4	p(¬d) = 0.7
p(e) = 0.7	
p(¬e) = 0.3	

Fig. 2. A: Example of CTG; B: Probabilities of condition outcomes

graphs where every condition, say a, has exactly two outcomes, $a = true$ or $a = false$. However, we can model multiple alternative outcomes, say $a = 1$ or $a = 2$ or $a = 3$ provided that they are mutually exclusive (i.e., only one of them is true at run time).

In Figure 2A t_0 is the root node and it is a fork node that always executes at run time. Arcs (t_0, t_1) and (t_0, t_{12}) rooted in an executing fork node are always evaluated to true. Node t_1 is a branch node, labeled with condition a. With an abuse of notation we have omitted the condition in the node and we have labeled arc (t_1, t_2) with the outcome a meaning $a = true$ and (t_1, t_5) with $\neg a$ meaning $a = false$. The probability of $a = true$ is 0.5 and the probability of $a = false$ is also 0.5.

Let $A^+(t_i)$ be the set of outgoing arcs of node t_i, that is $A^+(t_i) = \{a_k \in A \mid a_k = (t_i, t_j)\}$; similarly let $A^-(t_i)$ be the set of ingoing arcs of node t_i, i.e., $A^-(t_i) = \{a_k \in A \mid a_k = (t_j, t_i)\}$. Then t_i is a said to be a *root node* if $|A^-(t_i)| = 0$ (t_i has no ingoing arc), t_i is a *tail node* if $|A^+(t_i)| = 0$ (t_i has no outgoing arc).

Without loss of generality, we restrict our attention to CTGs such that every node t_i with two or more ingoing arcs ($|A^+(t_i)| > 1$) is either an *and-node* or an *or-node*. The concept of and/or-nodes, that of executing node and the arc status can be formalized in a recursive fashion:

Definition 2 *Run time execution of nodes, arc status, and/or node definition:*

- *The root node always executes.*
- *The status of arc (t_i, t_j) rooted in a fork node $t_i \in T_F$ is true if node t_i executes.*
- *The status of arc (t_i, t_j) rooted in a branch node $t_i \in T_B$ is true if node t_i executes and the outcome Out_{ij} labeling the arc is true.*
- *A node t_i with $|A^-(t_i)| > 1$ is an or-node if either none or only one of the*

ingoing arcs status can be true at run-time.

- *An or-node t_i executes if any arc in $A^+(t_i)$ has a status equal to true.*
- *A node t_i with $|A^-(t_i)| \geq 1$ is an and-node if it is possible that all the ingoing arcs status are true at run time.*
- *An and-node executes if all arcs in $A^-(t_i)$ have a status equal to true.*

Note that the definition is recursive: deciding whether a node t_i with $|A^+(t_i)|$ is an and/or-node depends on whether its predecessors can execute, and deciding whether a node can execute requires to know whether the predecessors are and/or-nodes. The system is however consistent as both concepts only depend on information concerning the *predecessors* of the considered node t_i; as the root node by definition always executes and the CTG contains no cycle, both the concept of and/or-node and that of executing node/status of an arc are well defined.

Observe that ingoing arcs in an or-node are always mutually exclusive; mixed and/or-nodes are not allowed but can be modeled by combining pure (possibly fake) and-nodes and or-nodes. Note also that nodes with a single ingoing arc are classified as and-nodes. Again, for the sake of simplicity in the paper we have used examples with only two ingoing arcs in and/or-nodes, but the presented results are valid in general and apply for any number of ingoing arcs.

In Figure 2A t_{15} is an or-node since at run time either the status of (t_{14}, t_{15}) or the one of (t_{13}, t_{15}) is true (depending on the outcome of condition d); t_{21} is an and-node since, if condition a has outcome *false*, arc (t_{20}, t_{21}) is true and arc (t_{10}, t_{21}) status is true if the outcome $c = true$ holds. Therefore, it is possible that both incoming arcs are true at run time. t_{15} executes if any of the ingoing arcs status is true, while t_{21} executes only if both the ingoing arc status evaluate to true.

4.1.1 Activation event of a node

For modeling purposes, it is useful to express the combination of outcomes determining the execution of a node as a compact expression. As outcomes are logical entities (either they are *true* or *false* at run time) it is convenient to formulate such combination of outcomes as a logical expression, referred here to as *activation event.*

The activation event of a node t_i is denoted as $\varepsilon(t_i)$ and can be obtained in a recursive fashion, similarly to definition of executing node and and/or-nodes. In practice:

$$\varepsilon(t_i) = \begin{cases} \texttt{true} & \text{if } |A^-(t_i)| = 0 \ (t_i \text{ is the root node}) \\ \bigvee_{a_k=(t_j,t_i)\in A^-(t_i)} \varepsilon(a_k) & \text{if } t_i \text{ is an or-node} \\ \bigwedge_{a_k=(t_j,t_i)\in A^-(t_i)} \varepsilon(a_k) & \text{if } t_i \text{ is an and-node or if } |A^-(t_i)| = 1 \end{cases}$$

and $\varepsilon(a_k)$ is the activation event of an arc a_k and is defined as follows:

$$\varepsilon(a_k = (t_i, t_j)) = \begin{cases} \varepsilon(t_i) & \text{if } t_i \text{ is a fork} \\ \varepsilon(t_i) \wedge Out_{ij} & \text{if } t_i \text{ is a branch} \end{cases}$$

For example, the activation event of task t_2 in Figure 2A is $\varepsilon(t_2) = a$, while the activation event of t_{21} is $\varepsilon(t_{21}) = ((\neg a \wedge b) \vee (\neg a \wedge \neg b)) \wedge (\neg a \wedge c) = (\neg a \wedge b \wedge c) \vee (\neg a \wedge \neg b \wedge c) = \neg a \wedge c \wedge (b \vee \neg b) = \neg a \wedge c$.

In general we need to express activation events in Disjunctive Normal Form (DNF), that is a disjunction of one or more conjunctions of one or more literals.

4.2 Control Flow Uniqueness

Even if many of the definitions and algorithms we present in this paper work in the general case, we are interested in specific CTG satisfying a property called *Control Flow Uniqueness* (CFU) [1]. Intuitively, CFU is satisfied if no node t_i in the graph requires for its execution the occurrence of two outcomes found on separated paths from the root to t_i. More formally:

Definition 3 *A CTG satisfies the CFU if for each and-node t_i, there is a single arc $a \in A^-(t_i)$ such that, for all other incoming arcs $a' \in A^-(t_i)$:*

$$\text{status of arc } a \text{ is true} \Rightarrow \text{status of arc } a' \text{ is true}$$

where the symbol "⇒" denotes the logical implication. Intuitively a single ingoing arc $a \in A^-(t_i)$ is *logically* responsible of the execution of the and-node t_i; if the status of such arc becomes true at some point of time, the status of all other ingoing arcs will become (or have become) true as well. Note the actual run time execution of t_i only occurs once all ingoing arcs have become true.

[1] In the rest of the paper, we will explicitly underline which algorithms/properties need the CFU.

As a consequence there is also only one path from the root to the and-node that is *logically* responsible for the execution of that node. More formally:

Corollary: If a CTG satisfies CFU, then for each task t_i each conjunction of condition outcomes in its activation event $\varepsilon(t_i)$ (in DNF) can be derived by collecting condition outcomes following a single path from the root node to t_i.

For example in Figure 3A, task t_8 is an and-node; its activation event is $\varepsilon(t_8) = (a \wedge b) \vee (\neg a \wedge b) = b$, thus CFU holds. Conversely, in Figure 3B both $\neg a$ and b are strictly required for the execution of t_7 and they do not appear in sequence along any path from the root to t_7; hence CFU is not satisfied.

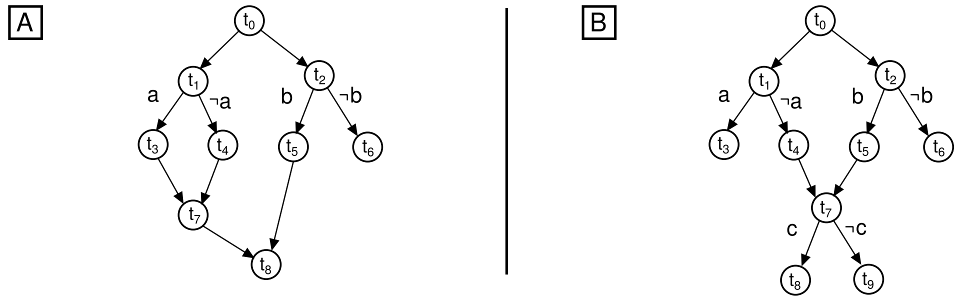


Fig. 3. A: a CTG which satisfies CFU — B: a CTG which does not satisfy CFU

In many practical cases CFU is not a restrictive assumption; for example, when the graph results from the parsing of a computer program written in a high level language (such as C++, Java, C#) CFU is quite naturally enforced by the scope rules of the language, or can be easily made valid by proper modeling. For example, consider again the pseudo-code in Figure 1. One can easily check that 1) CFU is satisfied and 2) there exist no simple translation of the pseudo code violating CFU as each conditional instruction (if) has a collector node (end if).

Moreover, in some application domains (e.g., business process management, embedded system design), a common assumption is to consider so called *structured* graphs, i.e., graphs with a single collector node for each conditional branch. In this case, the CFU is trivially satisfied. Note how a structured graph cannot model early exits (e.g. in case of error), as the one reported in Figure 1.

4.3 Sample Space and Scenarios

On top of a CTG we define the *sample space* S.

Definition 4 *The sample space of a CTG is the set of events occurring during all possible executions of a CTG, each event being a set of condition outcomes.*

For example, the sample space defined on top of the CTG in Figure 2A can be computed by enumerating all possible graph executions and contains 20 events. Again using an abuse of notation we refer to the outcome $a = true$ with a and to the outcome $a = false$ with $\neg a$. Also, for sake of clarity we have removed the logical conjunctions among conditions: the term $a \wedge b \wedge e$ has been simplified in abe. Therefore, the sample space associated to the CTG in Figure 2A is the following.

$$\begin{aligned}\mathcal{S} = \{&ade, ad\neg e, a\neg de, a\neg d\neg e, \neg abcde, \neg abc\neg de, \neg abcd\neg e,\\ &\neg abc\neg d\neg e, \neg a\neg bcde, \neg a\neg bc\neg de, \neg a\neg bcd\neg e, \neg a\neg bc\neg d\neg e,\\ &\neg ab\neg cde, \neg ab\neg c\neg de, \neg ab\neg cd\neg e, \neg ab\neg c\neg d\neg e, \neg a\neg b\neg cde,\\ &\neg a\neg b\neg c\neg de, \neg a\neg b\neg cd\neg e, \neg a\neg b\neg c\neg d\neg e\}\end{aligned}$$

We need now to associate a probability to each element of the sample space.

$$\forall s \in S \quad p(s) = \prod_{Out_{ij} \in s} p_{ij}$$

For instance, with reference to Figure 2A, the probability of event ade is $0.5 * 0.3 * 0.7 = 0.105$.

Each event in the sample space of the CTG is associated to a scenario. A scenario corresponds to a deterministic task graph containing the set of nodes and arcs that are active in the scenario. We have to define how to build such a task graph. This task graph is defined recursively.

Definition 5 *Given a CTG=$\langle T, A, C, P\rangle$, and an event $s \in S$ the deterministic task graph TG(s) associated with s is defined as follows:*

- *The CTG root node always belongs to the TG(s)*
- *A CTG arc (t_i, t_j) belongs to TG(s) if either*
  - · *t_i is a fork node and t_i belongs to TG(s) or*
  - · *t_i is a branch node, $Out_{ij} \in s$ and t_i belongs to TG(s).*
- *A CTG node t_i belongs to TG(s) if it is an and-node and all arcs $a_k \in A^-(t_i)$ are in TG(s) or if it is an or-node and any arc $a_k \in A^-(t_i)$ is in TG(s)*

TG(s) is called scenario associated with the event s.

With an abuse of notation, in the following we refer to the event s also as scenario. The deterministic task graph derived from the CTG in Figure 2A associated to the run time scenario $a = true$, $d = true$ and $e = false$ (or equivalently $ad\neg e$) is depicted in Figure 4.

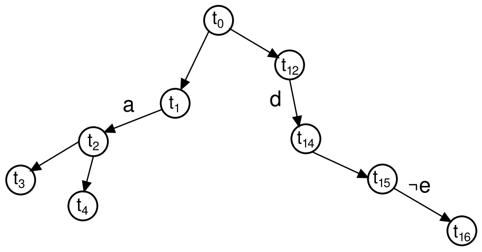


Fig. 4. The deterministic task graph associated with the run time scenario $a = true$, $d = true$ and $e = false$, for the CTG of Figure 2

.

Often we are interested in identifying a set of scenarios, such as for instance all scenarios where a given task executes. We have to start by identifying the events associated to scenarios where task t_i executes. This set is defined as $S_i = \{s \in S | t_i \in TG(s)\}$. The probability that a node t_i executes (let this be $p(t_i)$) can then be computed easily: $p(t_i) = \sum_{s \in S_i} p(s)$. For example, let us consider task t_2 in Figure 2A; then $S(t_2) = \{ade, ad\neg e, a\neg de, a\neg d\neg e\}$ and $p(t_2) = 0.5{\cdot}0.3{\cdot}0.7{+}0.5{\cdot}0.3{\cdot}0.3{+}0.5{\cdot}0.7{\cdot}0.7{+}0.5{\cdot}0.7{\cdot}0.3 = 0.5$. Alternatively, the probability $p(t_i)$ can be computed starting from the activation event; for example, $\varepsilon(t_2) = a$, hence $p(t_2) = p(a) = 0.5$, or $\varepsilon(t_8) = \neg a \wedge b$, hence $p(t_8) = p(\neg a) \cdot p(b) = 0.5 \cdot 0.4 = 0.2$.

For modeling purposes, we also define for each task an activation function $f_{t_i}(s)$; this is a stochastic function $f_{t_i} : S \rightarrow \{0, 1\}$ such that

$$f_{t_i}(s) = \begin{cases} 1 & \text{if } t_i \in TG(s) \\ 0 & \text{otherwise} \end{cases}$$

Finally, we need to define the concept of mutually exclusive tasks:

Definition 6 *Two tasks t_i and t_j are said to be mutually exclusive $mutex(t_i, t_j)$ iff there is no scenario $TG(s)$ where both tasks execute, i.e., where $t_i \in TG(s)$ and $t_j \in TG(s)$ or equivalently, $f_{t_i}(s) = f_{t_j}(s) = 1$.*

4.4 Scheduling and Allocation of a CTG: Problem definition and model

The allocation and scheduling problem we face is defined on a conditional task graph whose nodes are interpreted as activities (also referred to as tasks) and arcs are precedence relations between pairs of activities. The CTG is annotated with a number of activity features, such as duration, due and release dates, alternative resource sets and resource consumption. We have to schedule tasks and assign them resources from the alternative resource set such that all temporal and resource constraints in any run time scenario are satisfied and the expected value of a given objective function is optimized. More formally:

Definition 7 *An instance of the CTG allocation and scheduling problem is a tuple* $\langle CTG, Obj, Dur, Rel, Due, ResSet, ResCons\rangle$.

In the $CTG = \langle T, A, C, P\rangle$ T *represents the set of non preemptive tasks to be allocated and scheduled,* A *represents the set of precedence constraints between pairs of tasks,* C *is the set of conditions labeling the nodes and* P *is the set of outcomes and probabilities labeling the arcs.* Obj *is the objective function.* Dur, Rel, Due, $ResSet$ *and* $ResCons$ *are functions mapping each task in* T *to the respective duration, release date, due date, alternative resource set and resource consumption. Given a task* $t_i \in T$*, its duration is referred to as* Dur_i*, its release date as* Rel_i*, its due date as* Due_i*, its alternative resource set to as* $ResSet_i$ *and its resource consumption* $ResCons_i$*; with the exception of* $ResSet$ *all mentioned functions have values in* $\mathbb{N}^+$.

For sake of simplicity, we assume each task t_i needs a single resource taken from its $ResSet_i$ for its execution; however, the results presented in this paper can be easily extended to tasks requiring more than one resource. More in detail, suppose each task requires up to m of *types* of resource; provided separate $ResSet$ and $ResCons$ functions, all the presented results apply to each type in a straightforward fashion.

4.4.1 Modeling tasks and temporal constraints

As far as the model is concerned, each node in the CTG corresponds to a task (also called activity). Similarly to constraint-based scheduling, a task t_i is associated to a time interval $[start(t_i), end(t_i))$ where $start(t_i)$ is a decision variable denoting the starting time of the task and $end(t_i)$ is a variable linked to $start(t_i)$ as follows: $end(t_i) = start(t_i) + Dur_i$. Depending on the problem, the duration may be known in advance or may be a decision variable. In this paper we consider fixed, known in advance, durations.

The start time variable of a task t_i has a domain of possible values ranging from the earliest start time $est(t_i)$ to the latest start time $lst(t_i)$. Similarly, the end time variable has a domain ranging from earliest to the latest end times, referred to as $eet(t_i)$ and $let(t_i)$. Initially $start(t_i)$ and $end(t_i)$ range on the whole schedule horizon (from the time point 0 to the end of horizon $[0..eoh]$).

Each arc (t_i, t_j) in the CTG corresponds to a precedence constraint on decision variables and has the form $start(t_i) + Dur_i \leq start(t_j)$. Due dates and release dates translate to constraints $start(t_i) + Dur_i \leq Due_i$ and $start(t_i) \geq Rel_i$.

4.4.2 Modeling alternative resources

Beside start time of activities, an additional decision variable $res(t_i)$ represents the resource assigned to the activity t_i. The domain of possible values of $res(t_i)$ is $ResSet_i$.

Resources in the problem can be either discrete or unary. Discrete resources (also referred to as cumulative resources) have a known maximal capacity. A certain amount of resource $ResCons_i$ is consumed by activity t_i assigned to a discrete resource $res(t_i)$ at the start time of activity t_i and the same quantity is released at its end time.

A unary resource has a unit capacity. It imposes that all activities requiring the same unary resource are totally ordered.

Given a resource R its capacity is referred to as $Cap(R)$.

4.4.3 Classical objective function types

Depending on the problem, the objective function may depend on the temporal allocation of the activities, i.e., decisions on variables *start* (or equivalently variables *end* if the duration is fixed), or on the resource assignments, i.e., decisions on variables *res*.

In constraint-based scheduling, a widely studied objective function is the makespan, i.e., the length of the whole schedule. It is the maximum value of the $end(t_i)$ variables.

$$Obj_1 = \max_{t_i \in T} end(t_i) \tag{1}$$

Another example of objective function is the sum of costs of resource assignments to single tasks. As an example, consider the case where running a task on a given resource consumes a certain amount of energy or power.

$$Obj_2 = \sum_{t_i \in T} cost(t_i, res(t_i)) \tag{2}$$

In the hypothesis to have a cost matrix where each element c_{ij} is the cost of assigning resource j to task t_i, $res(t_i) = j \leftrightarrow cost(t_i, res(t_i)) = c_{ij}$.

A third example that we will consider in this paper still depends on resource assignment, but on pairs of assignments.

$$Obj_3 = \sum_{a_k=(t_i,t_j)\in A} cost(t_i, res(t_i), t_j, res(t_j)) \qquad (3)$$

For instance, suppose that arcs represent communication activities between two tasks. If t_i and t_j are assigned to the same resource, their communication cost is zero, while if they are assigned to different resources, the communication cost increases. Suppose we have a vector where each element c_k is the cost of arc $arc_k = (t_i, t_j)$ if t_i and t_j are assigned to different resources.

$$cost(t_i, res(t_i), t_j, res(t_j)) = \begin{cases} c_k \text{ if } res(t_i) \neq res(t_j) \\ 0 \text{ otherwise} \end{cases}$$

Other objective functions could be considered as well. For example there could be a cost for having at least one task using a certain resource (e.g. to turn the resource "on"). In this case the cost is associated to the execution of *sets* of tasks; the objective function can be considered a generalization of Obj_3 and is dealt with by means of the same techniques.

Clearly, having probabilities and conditional branches, we have to take into account all possible run time scenarios and optimize the expected value of the objective function. Therefore, given the objective function Obj and a scenario s, we refer to the objective function computed in the scenario s to as $Obj^{(s)}$. For example, in $Obj_1^{(s)}$, the maximum of end variables is restricted only to those tasks that are active in the scenario s (those belonging to TG(s)).

$$Obj_1^{(s)} = \max_{t_i \in TG(s)} end(t_i) = \max_{t_i \in T} f_{t_i}(s) end(t_i)$$

Similarly $Obj_2^{(s)} = \sum_{t_i \in TG(s)} cost(t_i) = \sum_{t_i \in T} f_{t_i}(s) cost(t_i, res(t_i))$

and

$$\begin{aligned} Obj_3^{(s)} &= \sum_{a_k=(t_i,t_j)\in TG(s)} cost(res(t_i), res(t_j)) = \\ &= \sum_{a_k=(t_i,t_j)\in A} f_{t_i}(s) f_{t_j}(s) cost(t_i, res(t_i), t_j, res(t_j)) \end{aligned}$$

Finally, by recalling the definition of "expected value" in probability theory, we state that:

Definition 8 *The expected value of a given objective function Obj is a weighted sum of the $Obj^{(s)}$ weighted by scenario probabilities*

$E(Obj) = \sum_{s\in S} p(s)Obj^{(s)}$

4.4.4 Solution of the CTG Scheduling Problem

The solution of the CTG scheduling problem can be given either in terms of a scheduling and allocation table [42] where each task is assigned to a different resource and a different start time, depending on the scenario, or as a unique allocation schedule where each task is assigned a single resource and a single start time independently on the run time scenario. The first solution is much more precise and able to better optimize the expected value of the objective function. Unfortunately, the size of such a table grows exponentially as the number of scenarios increases, making the problem of computing an optimal scheduling table P-SPACE complete (it is analogous to finding an optimal policy in stochastic constraint programming [40]). We therefore chose to provide a more compact solution, where each task is assigned a unique resource and a unique start time feasible in every possible run time scenario. this keeps the problem NP-hard. This choice goes in the line of the notion of strong controllability defined in [37] for temporal constraint networks with uncertainty; in particular, a network is said to be strongly controllable if there exists a single control sequence satisfying the temporal constraints for every scenario. In addition, for some classes of problems such as compilation of computer programs, this is the only kind of solution which can be actually implemented and executed [31]. More formally, we provide the following definition.

Definition 9 *The solution of the allocation and scheduling problem*

$$\langle\langle T, A, C, P\rangle, Obj, Dur, Rel, Due, ResSet, ResCons\rangle$$

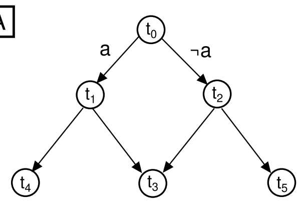
is an assignment to each task $t_i \in T$ *of a start time* $start(t_i) \in [0..eoh]$ *and of a resource* $res(t_i) \in ResSet_i$ *such that*

(1) $\forall t_i \in T \quad start(t_i) \geq Rel_i$
(2) $\forall t_i \in T \quad start(t_i) + Dur_i \leq Due_i$
(3) $\forall (t_i, t_j) \in A \quad start(t_i) + Dur_i \leq start(t_j)$
(4) $\forall t = 0 \ldots eoh \; \forall s \in S \quad \forall R \in \bigcup_{t_i \in TG(s)} ResSet_i :$

$$\sum_{\substack{t_i \in TG(s): \\ res(t_i) = R \\ start(t_i) \leq t < end(t_i)}} ResCons_i \leq Cap(R)$$

Constraints (1) and (2) ensure each task is executed within its release date and due date. Constraints (3) enforce precedence constraints, constraints (4) enforce resource capacity restrictions in all scenarios and at every time instant t on the time line. A solution is *optimal* if $E(Obj)$ is minimized (resp. maximized).

A



scenarios:

a	t_0 t_1 t_3 t_4
¬a	t_0 t_2 t_3 t_5

resource capacity:

$Cap(R_0) = 1$

$Cap(R_1) = 1$

all res requirements: 1

durations:	**start times:**
$Dur_0 = 4$	$start(t_0) = 0$
$Dur_1 = 3$	$start(t_1) = 4$
$Dur_2 = 4$	$start(t_2) = 4$
$Dur_3 = 3$	$start(t_3) = 8$
$Dur_4 = 3$	$start(t_4) = 7$
$Dur_5 = 3$	$start(t_5) = 8$

res sets:	**assigned res:**
$ResSet_0 = R_0, R_1$	$res(t_0) = R_0$
$ResSet_1 = R_0, R_1$	$res(t_1) = R_0$
$ResSet_2 = R_0, R_1$	$res(t_2) = R_0$
$ResSet_3 = R_0, R_1$	$res(t_3) = R_0$
$ResSet_4 = R_0, R_1$	$res(t_4) = R_1$
$ResSet_5 = R_0, R_1$	$res(t_5) = R_1$

B

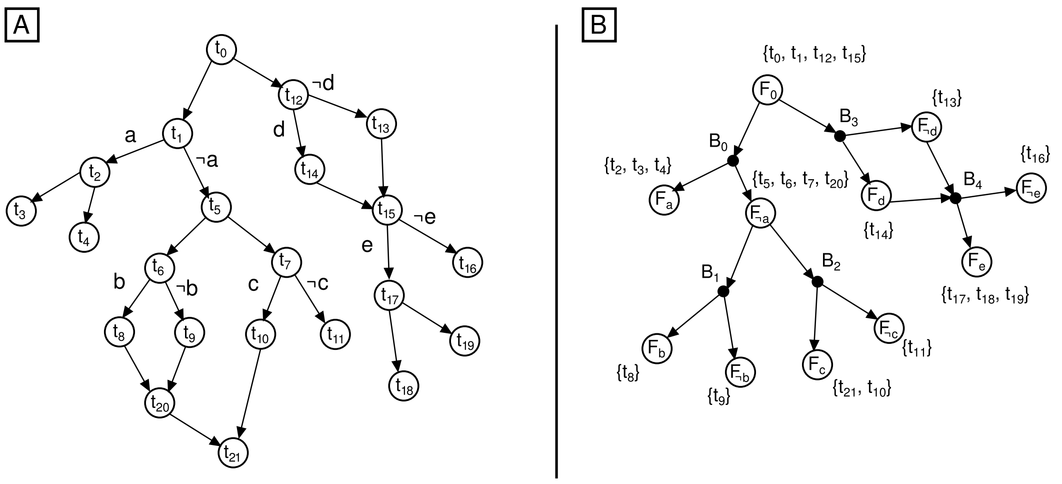
Fig. 5. A: a CTG scheduling problem; B: a possible solution

For example, in Figure 5A we show a small CTG scheduling problem and in Figure 5B the corresponding solution. Note that all tasks have a unique start time and a unique resource assignment independent from the scenario, but feasible in all scenarios. For instance, tasks t_1 and t_2 are mutually exclusive, as they cannot appear in the same scenario. Therefore, although they use the same unary resource, their execution can overlap in time.

5 Probabilistic Reasoning

The model presented in Section 4 cannot be solved via traditional constraint-based scheduling techniques. In fact, there are two aspects that require probabilistic reasoning and should be handled efficiently: the resource constraints to be enforced in all scenarios and the computation of the expected value of the objective function (a weighted sum on all scenarios). In both cases, in principle, we should be able to enumerate all possible scenarios, whose number is exponential. Thus, we need a more efficient way to cope with this expression.

One contribution of this paper is the definition of a data structure, called Branch/Fork Graph (BFG), that compactly represents all scenarios, and one parametric polynomial time algorithm working on the BFG that enables efficient probabilistic reasoning. For instance, we instantiate the parametric algorithm for the computation of the probability of a given set of scenarios, such as the probability of all scenarios where a given set of tasks execute (resp. do not execute).



C

p(a) = 0.5	p(b) = 0.4	p(c) = 0.6	p(d) = 0.3	p(e) = 0.7
p(¬a) = 0.5	p(¬b) = 0.6	p(¬c) = 0.4	p(¬d) = 0.7	p(¬e) = 0.3

Fig. 6. A: The CTG from Figure 2; B: The associated BFG; C: probabilities of condition outcomes

5.1 Branch/Fork Graph

A Branch/Fork Graph (BFG) intuitively represents the skeleton of all possible control flows and compactly encodes all scenarios of the corresponding CTG; for example Figure 6B shows the BFG associated to the CTG from Figure 2A (reported again in Figure 6A for simplicity).

A BFG is an acyclic directed graph. Nodes are either branch nodes (B-nodes, dots in Figure 6B) or fork nodes (F nodes, circles in Figure 6B). There is a branch node in the BFG for each branch node in the CTG. F-nodes instead represent sets of events and group CTG nodes executing at all such events. For example in Figure 6B F_a groups together nodes t_2, t_3 and t_4 as they all execute in all scenarios where $a = true$.

More formally:

Definition 10 *A Branch/Fork graph is a directed, acyclic graph associated to a CTG* $= \langle T = T_B \cup T_F, A, C, P\rangle$ *with two types of nodes, referred to as B-nodes and F-nodes.*

BFG nodes *satisfy the following conditions:*

(1) The graph has a B-node B_i for every branch node $t \in T_B$ in the associated CTG.

(2) Let S be the sample space of the CTG; then the BFG has an F-node F_i for every subset of events $\sigma \in 2^S$, unless:

(a) $\nexists t_i \in T$ such that $\forall s \in \sigma : \ t_i \in TG(s)$.

(b) $\exists c_k \in C$ such that (I) more than one outcome of c_k appear in scenarios in σ and (II) not all the outcomes of c_k appear in σ.
(c) $\exists \sigma' \in 2^S$ such that (a) and (b) are not satisfied (i.e. hence, an F-node would be built) and $\sigma \subset \sigma'$.

The CTG branch node corresponding to a B-node B_i is denoted as $t(B_i)$. The F-nodes are said to "represent" the set of events σ they correspond to, denoted as $\sigma(F_i)$. The set of CTG nodes t_i such that $\forall s \in \sigma(F_i)$, $t_i \in TG(s)$ is said to be "mapped on" F_i and is denoted as $t(F_i)$ [2]

BFG arcs *satisfy the following conditions:*

(1) An F-node F_i is connected to a B-node B_j by an arc (F_i, B_j) if $t(B_j) \in t(F_i)$
(2) A B-node B_i is connected by means of an arc labeled with the outcome $Out_{t(B_i),k}$ to an F-node F_j such that $t_k \in t(F_j)$
(3) An F-node F_i is connected to an F-node F_j such that no path from F_i to F_j already exists and:
(a) $\sigma(F_j) \subset \sigma(F_i)$
(b) There exists no F-node F_k such that $\sigma(F_j) \subset \sigma(F_k) \subset \sigma(F_i)$

Condition on BFG nodes (1) tells the BFG has one B-node for each branch in the associated CTG. Following condition (2), each F-node models a subsets of events $\sigma \in 2^S$; there is however no need to model a subset σ if any of the three conditions (2abc) holds. In particular:

2a) there is no need to model sets of events σ such that no task in the graph would be mapped to the resulting F-node; such sets of events are of no interest, as the ultimate purpose of the BFG is to support reasoning about task executions and their probability.
2b) there is no need to model a set of events σ, if two or more outcomes of a condition c_k appear in σ and still there is some outcome of c_k not in σ. In fact if two or more (but not all) outcomes of c_k are in σ, then σ still *depends* on c_k and one could model this by using several F-nodes, each one referring to a single outcome. If however all outcomes of c_k are in σ, then σ is *independent* on c_k.
2c) provided neither condition (2a) nor (2b) holds, there is still no need to build an F-node if there exist a larger set of events σ' such that neither condition (2a) nor (2b) holds as well. In practice, due to condition (2c) F-nodes always model maximal sets of events.

For instance, according to the definition, the BFG corresponding to the CTG in Figure 6A contains a branch node for each branch: B_0 corresponds to t_1 (i.e. $t(B_0) = t_1$), B_1 to t_6, B_2 to t_7, B_3 to t_{12}, B_4 to t_{15}. As for F-nodes, F_0 represents

[2] Note that $t(B_i)$ is a node, while $t(F_i)$ is a set.

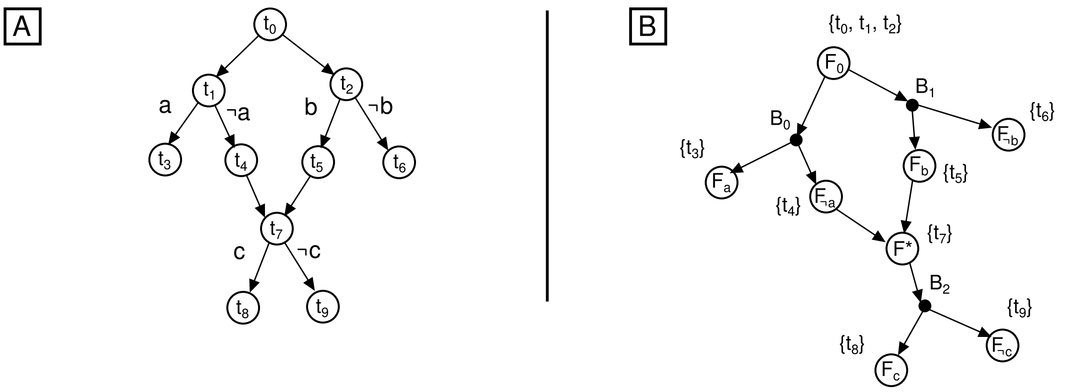


Fig. 7. A: The non CFU CTG from Figure 3.B; B: The associated BFG

the whole sample space and nodes t_0, t_1, t_{12}, t_{15} are mapped to it (i.e. $t(F_0) = \{t_0, t_1, t_{12}, t_{15}\}$), as they execute in all scenarios. F-node F_b corresponds to the set of events $\{\neg abcde, \neg abc\neg de, \neg abcd\neg e, \neg abc\neg d\neg e, \neg ab\neg cde, \neg ab\neg c\neg de,$ $\neg ab\neg cd\neg e, \neg ab\neg c\neg d\neg e\}$, that is the set of events where outcomes $\neg a$ and b are both true; t_8 is the only task in $t(F_b)$ as it executes in all such scenarios (condition (2b)) and does not execute in any superset of scenarios (condition (2c)).

Concerning the BFG connectivity, condition (1) intuitively states that every B-node has an ingoing arc from all F-nodes where the corresponding CTG branch is mapped; in the BFG of Figure 6B condition (1) yields all arcs from F-nodes to B-nodes. Condition (2) defines instead the connectivity from B-nodes to F-nodes: it tells that every B-node has an outgoing arc for each outcome of the corresponding CTG branch; the destination of such BFG arc is the F-node where the destination of the arc with that outcome (task t_k) is mapped to. The BFG arc is labeled with the corresponding CTG outcome. In Figure 6B condition (2) yields all arcs from B-nodes to F-nodes.

Finally condition (3) (that never happens in CTG satisfying the CFU) defines connectivity between F-nodes and other F-nodes linked by no path resulting from conditions (1) and (2). In particular, arcs (F_i, F_j) are built where F_j is the destination F-node, and F_i is the “minimal” (see condition (3b)) F-node such that $\sigma(F_j) \subset \sigma(F_i)$ (see condition (2b)). Observe that parents of B-nodes are always F-nodes, parents of F-node can be both F-nodes and B-nodes; children of B-nodes are always F-nodes, children of F-nodes can be both F-nodes and B-nodes. As an example consider Figure 7 where we have links between F-nodes in the BFG corresponding to a CTG that does not satisfy CFU.

Some properties follow from the BFG definition. First of all, *given a CTG, its associated BFG is uniquely defined.* The result comes from the fact that node condition (1) univocally defines the set of B-nodes, node condition (2c) univocally selects the set of scenarios to which every F-node corresponds to and the graph connectivity is univocally defined once F-nodes and B-nodes are given.

The family of mappings of CTG nodes to F-nodes in general does not partition nodes in the CTG; Figure 8 shows an example graph where a node (namely t_3) is mapped to more than one F-node in the BFG ($F_{\neg a}, F_{\neg b}$). The following theorem holds:

Theorem 1 *If a CTG node t_i is mapped to more than one F-node, such F-nodes are mutually exclusive, and represent pairwise disjoint sets of scenarios.*

Proof: Suppose a t_i is both mapped on F-nodes F_j and F_h; this can be true only if no set of events σ' exists such that (2a), (2b) (2c) are satisfied. From condition (2b), $t_i \in TG(s) \forall s \in \sigma(F_j) \cup \sigma(F_h)$, hence $\sigma' = \sigma(F_j) \cup \sigma(F_h)$ satisfies (2b), and of course (2c). Hence $\sigma' = \sigma(F_j) \cup \sigma(F_h)$ must violate (2a), or t_i would not be mapped to F_j and F_h. Therefore, there must be a condition c_k (let O be the set of outcomes) such that events in $\sigma(F_j) \cup \sigma(F_h)$ do not contain the whole set of outcomes of c_k and $\sigma(F_j)$, $\sigma(F_h)$ contains exactly one outcome of c_k. As different outcomes of the same condition generate mutually exclusive events, $\sigma(F_j)$ and $\sigma(F_h)$ are mutually exclusive. $\square$

Note also that in general F-nodes and B-nodes can have more than one parent (despite this is not the case for F-nodes in Figure 6 and 8), as well as more than one child. In particular:

Theorem 2 *Parents of B-nodes are always mutually exclusive; parents of F-nodes are never mutually exclusive.*

Proof: Let B_i be a B-node; due to connectivity condition (1), its parents are the F-nodes where the corresponding CTG branch $t(B_i)$ is mapped; due to Theorem 1 those F-nodes are mutually exclusive.
Now, let F_i be an F-node, with F-node parents F' and F'', then $\sigma(F_i) \subset \sigma(F')$ and $\sigma(F_i) \subset \sigma(F'')$, due to connectivity condition (3). Note that the strict inclusion holds, hence $\sigma(F') \nsubseteq \sigma(F'')$, $\sigma(F'') \nsubseteq \sigma(F'')$ and $\sigma(F') \cap \sigma(F'') \neq \emptyset$. Therefore the two parents are non mutually exclusive, as they share some event. The reasoning still holds when one parent (or both) is a B-node B_j, by substituting $\sigma(F')$ with $\{s \in \sigma(F') \mid t(B_j) \in t(F')\}$. $\square$

5.2 BFG and Control Flow Uniqueness

Control flow uniqueness translates into additional properties for the BFG:

Theorem 3 *If CFU holds, every F-node has a single parent and it is always a B-node.*

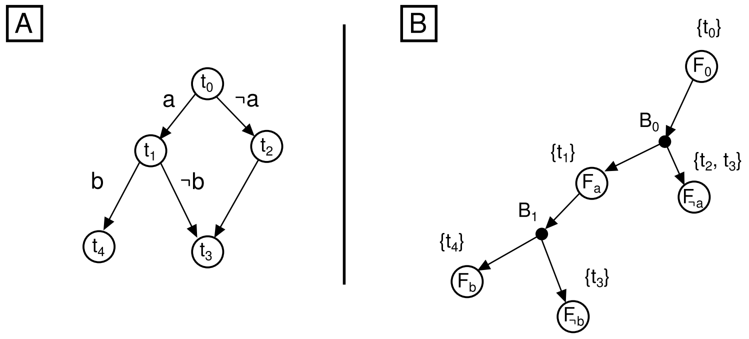


Fig. 8. Task t_3 is mapped on two F-nodes $(F_{\neg a}, F_{\neg b})$

Proof.
[Every F-node has a single parent]
Suppose a node F_i has two parents F', F'' (see the proof of Theorem 2 for how to adapt the reasoning to B-nodes).
Let t_j be a CTG and-node $\in t(F_i)$, and consider two incoming arcs $a' = (t', t_j)$ and $a'' = (t'', t_j)$ such that $a' \not\Rightarrow a''$ (for CFU to hold); either the parents t' and t'' are in $t(F_i)$ as well, or they are mapped to some *ancestors* of F_i; in this case they execute in the events represented by any descendant of such ancestors (as a consequence of connectivity conditions), hence they also execute on some parents of F_i.
It is therefore always possible to identify a t_j such that:
(a) $t_j \in t(F_i)$
(b) parents t' and t'' respectively execute in all events in $\sigma(F')$ and $\sigma(F'')$.

Now, since $a' \not\Rightarrow a''$, in terms of set of events this implies $\sigma(F'') \subset \sigma(F')$. However, due to Theorem 2, we know that and $\sigma(F'') \not\subseteq \sigma(F')$, which leads to a contradiction. Hence, if CFU holds, every F-node has a single parent. $\square$

[The parent is a B-node]
Suppose there exists an F-node F_i with a single, F-node, parent F'. As F' is the only parent of F_i and there is no intermediate B-node, every node $t_j \in t(F_i)$ executes in $\sigma(F')$ as well. At the same time, due to connectivity condition (3), $\sigma(F_i) \subset \sigma(F')$; hence such a node F_i, if existent, would fail to satisfy node condition (2c). Therefore, the single parent of every F-node is a B-node. $\square$

From Theorem 3 we deduce that if CFU holds the BFG is a bi-chromatic *alternate* graph. Moreover, since every branch node with m outgoing arcs originates exactly m F-nodes, the BFG has exactly $n_o + 1$ F-nodes, where n_o is the number of condition outcomes; for this reason, when CFU is satisfied, one can denote F-nodes (other than the root) by the outcome they refer to; for example an F-node referring to outcome $a = true$ will be denoted as F_a, if referring to $b = false$ as $F_{\neg b}$ and so on.

CFU is also a necessary condition for the structural property listed above to hold; therefore we can check CFU by trying to build a BFG with a single parent for each F-node: if we cannot make it, then the original graph does not satisfy the condition. The BFG construction procedure in case CFU is satisfied is outlined in Appendix A.

5.3 BFG and scenarios

The most interesting feature of a BFG is that it can be used to select and encode groups of scenarios in which arbitrarily chosen nodes execute. A specific algorithm can then be applied to such scenarios, in order to compute the corresponding probability, or any other feature of interest.

Groups of scenarios are encoded in the BFG as sets of s-trees:

Definition 11 (s-tree) *An s-tree is any subgraph of the BFG satisfying the following properties:*

(1) The subgraph includes the root node
(2) If the subgraph includes an F node, it includes also all its children
(3) If the subgraph includes an F-node, it includes also all its parents
(4) If the subgraph includes a B-node, it includes also one and only one of its children.

Note that the s-tree associated to a scenario s is the BFG associated to the deterministic task graph $TG(s)$.

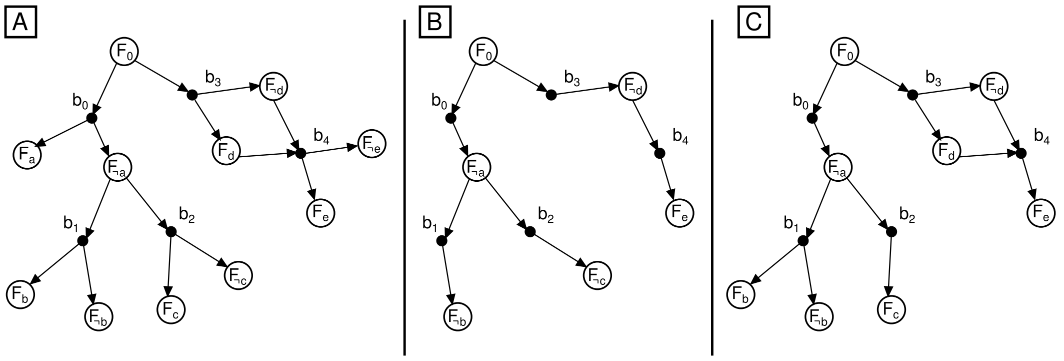


Fig. 9. (A) BFG for the graph in Figure 6 - (B) the s-tree associated to the scenario $\neg a \neg b \neg c \neg d e$ - (C) a subgraph (set of s-trees) associated to scenarios where $\neg a c e$ holds

Despite its name, an s-tree is not necessarily a tree: this is always the case only if CFU holds (which is not required by Definition 11) (see Figure 9A/B, where CFU holds and F-nodes are labeled with the condition outcome they refer to).

By relaxing condition (4) in Definition 11 and allowing the inclusion of more than one condition per B-node, we get a subgraph representing a set of s-trees; a single s-tree (and hence a scenario) can be derived by choosing from the subgraph a single outcome per branch condition. For example from the subgraph in Figure 9C one can extract the set of s-trees corresponding to $\neg abcde$, $\neg abc\neg de$, $\neg a\neg bcde$, $\neg a\neg bc\neg de$. This encoding method is sufficient to represent sets of scenarios of practical interest (e.g. those required by the algorithm and constraints discussed in the paper). The importance of s-trees mainly lies in the fact that they are required for the algorithm presented in the forthcoming Section 5.5.

5.4 Querying the BFG

We now restrict our attention to CTG satisfying Control Flow Uniqueness; namely we want to provide a way to select a set of s-trees representing a set of scenarios which include or exclude a specified group of nodes; once such a subgraph is available, the execution probability can be extracted by a proper algorithm. We consider selection rules specified by means of either conjunctions or disjunctions of positive and negative terms[3]. Each basic term of the query can be either t_i (with meaning "task t_i executes") or $\neg t_i$ (with meaning "task t_i does not execute"). Some examples of valid queries are:

$$q_0 = t_i \wedge t_j \qquad q_1 = t_i \wedge \neg t_j \wedge \neg t_k \qquad q_2 = t_i \vee t_j$$

A query returns the BFG subgraph representing the events where the specified tasks execute/do not execute, or *null* in case no such event exists. The idea of the query processing procedure is that, since the complete BFG represents all possible scenarios, we can select a subset of them by removing F-nodes which do not satisfy the boolean query. Thus, in order to be processed, queries are first negated:

$$\neg q_0 = \neg t_i \vee \neg t_j \qquad \neg q_1 = \neg t_i \vee t_j \vee t_k \qquad \neg q_2 = \neg t_i \wedge \neg t_j$$

Each element in the negated disjunction now has to be mapped to a set of F-nodes to be removed from the BFG. This can be efficiently done by pre-computing for each BFG node an *inclusion label* and an *exclusion label*:

[3] Mixed queries are also allowed by converting them to groups of conjunctive queries representing disjoint sets of scenarios, but paying an exponential complexity blow-up, depending on the size and the structure of the query. Pure conjunctive and disjunctive queries are however enough for managing cases of practical interest as shown in the rest of the paper.

1. Inclusion labels

A CTG task t_i is in the *inclusion label* $i(F_j)$ of an F-node F_j either if it is directly mapped on it, $t_i \in t(F_j)$, or if t_i is in the inclusion label of any of its parents.

A CTG task t_i is in the *inclusion label* $i(B_j)$ of a B-node B_j if t_i is in the inclusion label of all of its parents.

In practice, $t_i \in i(F_j)$ (resp. $i(B_j)$) if it *does* execute in all scenarios corresponding to every s-tree containing F_j (resp. B_j).

2. Exclusion labels

A CTG task t_i is in the *exclusion label* $e(F_j)$ of an F-node F_j either if parents of F_j are F-nodes[4] and t_i is in the exclusion label of any parent, or if the parent of F_j is a B-node and it exists a brother node F_k such that t_i is mapped on a descendant (either direct or not) of F_k and t_i is not mapped on a descendant (either direct or not) of F_j.

A CTG task t_i is in the *exclusion label* $e(B_j)$ of a B-node B_j if t_i is in the exclusion label of all of its parents.

In practice, $t_i \in e(F_j)$ (resp. $e(B_j)$) if it *cannot* execute in the scenario correspondent to any s-tree containing F_j (resp. B_j).

For example in Figure 6B (reproduced in Figure 10A for sake of clarity), the inclusion label of node F_0 is $i(F_0) = \{t_0, t_1, t_{12}, t_{15}\}$ and $i(B_3)$ is equal to $i(F_0)$; then $i(F_d) = i(F_0) \cup \{t_{14}\}$ and $i(F_{\neg d}) = i(F_0) \cup \{t_{13}\}$; $i(B_4)$ is again equal to $i(F_0)$, since neither t_{13} nor t_{14} are mapped on both parents of B_4. As for the exclusion labels: $e(F_0) = \emptyset$ and $e(B_0) = \emptyset$; $e(F_{\neg a}) = \{t_2, t_3, t_4\}$, since those tasks are mapped on the brother node F_a and they are not mapped on any descendant of $F_{\neg a}$.

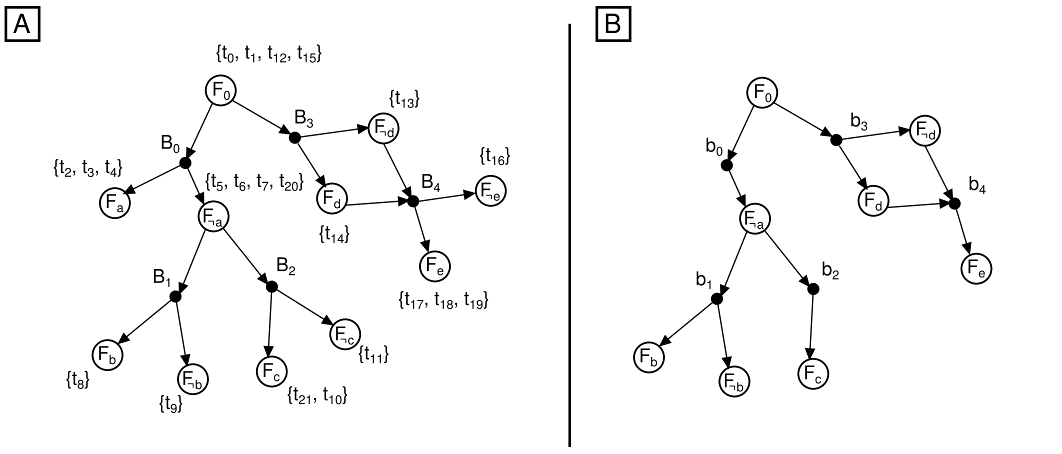
Once inclusion and exclusion labels are computed, each (conjunctive) term of the query (e.g. $t_i \wedge \neg t_j \wedge \ldots$) is mapped to a set of F/B-nodes satisfying $t_i \in i(F_j)$ (or $i(B_j)$) for every positive element t_i in the term, and $t_i \in e(F_j)$

[4] This property holds for general CTG, while if CFU is satisfied parents of F-nodes are always B-nodes.

(or $e(B_j)$) for each negative element $\neg t_i$ in the term. For example:

$$\begin{aligned}
t_i &\rightarrow \{F_j \mid t_i \in i(F_j)\} \cup \{B_j \mid t_i \in i(B_j)\} \\
\neg t_i &\rightarrow \{F_j \mid t_i \in e(F_j)\} \cup \{B_j \mid t_i \in e(B_j)\} \\
t_i \wedge t_k &\rightarrow \{F_j \mid t_i, t_k \in i(F_j)\} \cup \{B_j \mid t_i, t_k \in i(B_j)\} \\
t_i \wedge \neg t_k &\rightarrow \{F_j \mid t_i \in i(F_j),\ t_k \in e(F_j)\} \cup \{B_j \mid t_i \in i(B_j),\ t_k \in e(B_j)\}
\end{aligned}$$

Note that an (originally) conjunctive query is mapped to a set of terms, each consisting of a single positive or negative task literal; the query is processed by removing from the complete BFG the F and B-nodes corresponding to each term. Conversely, an (originally) disjunctive query yields a single term consisting of a conjunction of positive of negative task; the query is processed by removing from the BFG the F and B-nodes corresponding to the term. For example, on the graph of Figure 6B (reproduced in Figure 10A for sake of clarity), the query $q = t_{21} \wedge \neg t_3 \wedge \neg t_{16} = \neg(\neg t_{21} \vee t_3 \vee t_{16})$ is processed by removing from the BFG $F_{\neg c}$, F_a and $F_{\neg e}$, since $t_{21} \in e(F_{\neg c})$, $t_3 \in i(F_a)$ and $t_{16} \in i(F_{\neg e})$. The resulting subgraph is the one shown in Figure 9C (reproduced in Figure 10B).



p(a) = 0.5	p(b) = 0.4	p(c) = 0.6	p(d) = 0.3	p(e) = 0.7
p(¬a) = 0.5	p(¬b) = 0.6	p(¬c) = 0.4	p(¬d) = 0.7	p(¬e) = 0.3

Fig. 10. (A) the BFG from Figure 6B (B) the subgraph from Figure 9C

Disconnected nodes are removed at the end of the process. During query processing, one has to check whether at some step any B-node loses all children; in such case the output is *null*, as the returned BFG subgraph would contain a B-node with no allowed outcome and this is impossible. Similarly, the result is *null* if all BFG nodes are removed. A query is always processed in linear time. Finally, the following theorem holds:

Theorem 4 *If a query returns a BFG subgraph, this represents a set of s-trees.*

Proof: Assume the query result is not *null* and remember we consider CFU to be satisfied; then condition (1) in Definition 11 is trivially satisfied, as a non-empty query always includes the root node. Conditions (2) and (3) are satisfied as, in a graph satisfying CFU, children and parents of F-nodes are always B-nodes, and B-nodes are never removed when processing a query. Finally, condition (4) is satisfied as query processing may remove some children of a B-node, but not all of them (or *null* would be returned). □

As the result of a query is always a set of s-trees, it can be used as input for the backward visit algorithm.

5.5 Visiting the BFG

Many algorithms along the paper are based on a backward visit of the BFG. During these visits each algorithm collects some attributes stored in F and B nodes. We therefore propose a meta algorithm, using a set of parameters which have to be defined case by case. All backward visit based algorithms assume CFU is satisfied and require as input a subgraph representing a set of s-trees (hence a BFG as a particular case).

In particular, Algorithm 1 shows the generic structure of a backward visit of a given BFG. The visit starts from the leaves and proceeds to the root; every predecessor node is visited when all its successors are visited (lines 12-13). The meta algorithm is parametric in the five-tuple $\langle A, init_F, init_B, update_F, update_B\rangle$. In particular A is a set of possible attribute values characterizing each F and B-node, and $A(n)$ denotes the values of the attributes for node n; the function $init_F : \{F_i\} \to A$ associates to an F-node an element in A, that is values for each of its attributes, and the function $update_F : \{F_i\} \times \{B_j\} \to A$ associates to an F-node and a B-node an element in A. The functions $init_B : \{B_i\} \to A$ and $update_B : \{B_i\} \times \{F_j\} \to A$ are defined similarly to $init_F$ and $update_F$ for B-nodes. In the algorithm, $init_F$ and $init_B$ are used to assign an initial value of attributes in A to each F and B-node (line 2); the function $update_F$ is used to update the attribute values of the *parent* of an F-node (which is a B-node — line 6) and $init_B$ is used to update the attributes of the *parent* of a B-node (which is an F-node — line 8). In the following, we will use Algorithm 1 with different parameter settings for different purposes.

5.6 Computing subgraph probabilities

In the following we show how to compute the probability for a given BFG, or part of it (sets of s-trees derived from querying the BFG).

Algorithm 1 Backward visit(A, $init_F$, $init_B$, $update_F$, $update_B$)

```
 1: let $L$ be the set of nodes to visit and $V$ the one of visited nodes. Initially
    $L$ contains all subgraph leaves and $V = \emptyset$
 2: for each F and B-node, store the values of attributes in $A$. Initially set
    $A(n) = init_F(n)$ for all F nodes, $A(n) = init_B(n)$ for all B-nodes
 3: while $L \neq \emptyset$ do
 4:   pick a node $n \in L$
 5:   if $n$ is an F-node with parent $n_p$ then
 6:     $A(n_p) = update_F(n, n_p)$
 7:   else if $n$ is a B-node then
 8:     for every parent $n_p$: $A(n_p) = update_B(n, n_p)$
 9:   end if
10:   $V = V \cup \{n\}$
11:   $L = L \setminus \{n\}$
12:   for every parent $n_p$ do
13:     if all children of $n_p$ are in $V$ then $L = L \cup \{n_p\}$
14:   end for
15: end while
```

The probability of a subgraph can be computed via a backward visit which is an instantiation of the meta Algorithm 1. In particular, a single attribute p, representing a probability, is stored in F and B-nodes, and thus $A = [0, 1]$. The result of the algorithm is the probability value of the root node. The *init* and *update* functions are as follows:

$$\begin{aligned} init_F(F_i) &= \text{the probability of the outcome labeling the arc} \\ & \text{ from the single B-node parent of } F_i \text{ and } F_i \text{ itself} \\ init_B(B_i) &= 0 \\ update_F(F_i, B_j) &= p(B_j) + p(F_i) \\ update_B(B_i, F_j) &= p(F_j) \cdot p(B_i) \end{aligned}$$

As an example, consider the subgraph of Figure 9C (also reported in Figure 10B, together with the probabilities). The computation starts from the leaves; for example at the beginning $p(F_b) = 0.4$, $p(F_{\neg b}) = 0.6$, $p(F_c) = 0.6$ (set by $init_F$). Then, probabilities of B-nodes are the weighted sum of those of their children (see $update_F$); for example $p(b_1) = p(F_b)+p(F_{\neg b}) = 0.4+0.6 = 1$ and $p(b_2) = p(F_c) = 0.6$. Probabilities of F-nodes are instead the product of those of their children (see $update_B$), and so $p(F_{\neg a}) = p(b_1)p(b_2) = 0.6$. The visit proceeds backwards until $p(F_0)$ is computed, which is also the probability of the subgraph.

6 OBJECTIVE FUNCTION

One of the purposes of the probabilistic reasoning presented so far is to derive the expected value of a given objective function efficiently. We consider in this section three examples of objective functions that are commonly used in constraint based scheduling, described in Section 4.4.3: the minimization of costs of single task-resource assignments, the minimization of the assignment cost of pairs of tasks, and the makespan. We refer to the first two examples as *objective functions depending on the resource allocation* while we refer to the third case as *objective function depending on the task schedule.*

This first and the second case are easier since we can transform the expected value of the objective function in a deterministic objective function provided that we are able to compute the probability a single task executes and the probabilities that a pair of tasks executes respectively. The third example is much more complicated since there is not a declarative description of the objective function that can be computed in polynomial time. Therefore, we provide an operational definition of such expected value by defining an expected makespan constraint, and the corresponding filtering algorithm.

6.1 Objective function depending on the resource allocation

We first consider an objective function depending on single tasks assignments and on the run time scenario; for example, suppose there is a fixed cost for the assignment of each task t_i to a resource $res(t_i)$, as it is the case for objective (2) in Section 4.4.3. The general form of the objective function on a given scenario s is.

$$Obj^{(s)} = \sum_{t_i \in TG(s)} cost(t_i, res(t_i)) = \sum_{t_i \in T} f_{t_i}(s) cost(t_i, res(t_i))$$

We remind that $f_{t_i}(s) = 0$ if $t_i \notin TG(s)$. According to Definition 8, the expected value of the objective function is

$$E(Obj) = \sum_{s \in S} p(s) Obj^{(s)} = \sum_{s \in S} p(s) \sum_{t_i \in T} f_{t_i}(s) cost(t_i, res(t_i))$$

We remind that $S_i = \{s \mid t_i \in TG(s)\}$ is the set of all scenarios where task i executes. Thus,

$$E(Obj) = \sum_{t_i \in T} \left[cost(t_i, res(t_i)) \sum_{s \in S_i} p(s) \right]$$

Now every stochastic dependence is removed and the expected value is reduced to a deterministic expression. Note that $\sum_{s \in S_i} p(s)$ is simply the probability of execution of node/task i. This probability can be efficiently computed by running Algorithm 1 instantiated as explained in Section 5.6, on the BFG sub-graph resulting from the query $q = t_i$.

As a second example, we suppose the objective function is related to arcs and to the run time scenario; again, we assume there is a fixed cost for the activation of an arc, as it is the case for the objective (3) in Section 4.4.3. The general form of the objective function is.

$$\begin{aligned} Obj^{(s)} &= \sum_{a_k=(t_i,t_j) \in TG(s)} cost(res(t_i), res(t_j)) = \\ &= \sum_{a_k=(t_i,t_j) \in A} f_{t_i}(s) f_{t_j}(s) cost(t_i, res(t_i), t_j, res(t_j)) \end{aligned}$$

The expected value of the objective function is

$$E(Obj) = \sum_{s \in S} p(s) \sum_{a_k=(t_i,t_j) \in A} f_{t_i}(s) f_{t_j}(s) cost(t_i, res(t_i), t_j, res(t_j))$$

note that $cost(t_i, res(t_i), t_j, res(t_j))$ is a cost that we can derive from a cost matrix c

$$E(Obj) = \sum_{a_k=(t_i,t_j) \in A} \left[cost(t_i, res(t_i), t_j res(t_j)) \sum_{s \in S_i \cap S_j} p(s) \right]$$

Now every stochastic dependence is removed and the expected value is reduced to a deterministic expression. Note that $\sum_{s \in S_i \cap S_j} p(s)$ is the probability that both tasks i and j execute. Again this probability can be efficiently computed using Algorithm 1 on the BFG sub-graph resulting from query $q = t_i \wedge t_j$.

6.2 Objective function depending on the task schedule

For a deterministic task graph, the makespan is simply the end time of the last task; it can be expressed as: $makespan = \max\{end(t_i) \mid t_i \in T\}$. If the task graph is conditional the last task depends on the occurring scenario. Remember we are interested in finding a single assignment of start times, valid for each execution scenario; in this case each scenario s identifies a deterministic Task Graph ($TG(s)$) and its makespan is $\max\{end(t_i) \mid t_i \in TG(s)\}$. Thus, the most natural declarative expression for the expected makespan would be:

$$E(makespan) = \sum_{s \in S} p(s) \max\{end(t_i) \mid t_i \in TG(s)\} \tag{4}$$

where $p(s)$ is the probability of the scenario s. Note that the expression can be simplified by considering only tail tasks (i.e. tasks such that $|A^+(t_i)| = 0$). For example, consider the CTG depicted in Figure 11A, the scenarios are $\{a\}$, $\{\neg a, b\}$, $\{\neg a, \neg b\}$, and the expected makespan can be expressed as:

$$\begin{aligned} E(makespan) = \; & p(a) \max\{end(t_2), end(t_6)\} + \\ & p(\neg a \wedge b) \max\{end(t_4), end(t_6)\} + \\ & p(\neg a \wedge \neg b) \max\{end(t_5), end(t_6)\} \end{aligned}$$

Unluckily the number of scenarios is exponential in the number of branches, which limits the direct use of expression (4) to small, simple instances. Therefore, we defined an expected makespan global constraint

$exp\_mkspan\_cst([end(t_1), \ldots, end(t_n)], emkspan)$

whose aim is to compute legal bounds on the expected makespan variable $emkspan$ and on the end times of all tasks ($end(t_i)$) in a procedural fashion. We devised a filtering algorithm described in Section 6.2.1 whose aim is to prune the expected makespan variable on the basis of the task end variables, and vice-versa, see Section 6.2.2.

6.2.1 Filtering the expected makespan variable

The filtering algorithm is based on a simple idea: the computation of the expected makespan is tractable when the order of tasks, and consequently of end variables, is known. Consider the schedule in Figure 11B, where all tasks use a unary resource $URes_0$: since t_5 is the last task, the makespan of all scenarios containing t_5 is $end(t_5)$. Similarly, since t_4 is the last but one task, $end(t_4)$ is the makespan value of all scenarios containing t_4 and not containing t_5, and so on.

The computation can be done even if start times have not yet been assigned, as long as the end-order of tasks is known; in general, let $t_0, t_1, \ldots, t_{n_t-1}$ be the sequence of CTG tasks ordered by increasing end time, then:

$$E(makespan) = \sum_{i=0}^{n_t-1} p(t_i \wedge \neg t_{i+1} \wedge \ldots \wedge \neg t_{n_t-1})\ end(t_i) \tag{5}$$

The expected makespan can therefore be computed as a weighted sum of end times, where the weight of task t_i is given by the probability that 1) t_i executes 2) none of the task ending later than t_i executes. The sum contains n_t terms, where n_t is the number of tasks; again this number can be decreased by considering tail tasks only.

Hence, *once the end order of tasks is fixed*, we can compute the expected makespan in polynomial time, we just need to be able to efficiently compute the probability weights in expression (5): if CFU holds, this can be done as explained in Section 5 by running Algorithm 1 (in its probability computation version) on the BFG subgraph resulting from query $q = t_i \wedge \neg t_{i+1} \wedge \ldots \wedge \neg t_{n_t-1}$.

In general, however, during search the order of tasks is not fixed, but it is always possible to identify possibly infeasible task schedules whose makespan, computed with expression (5), can be used as a bound for the expected makespan variable. We refer to these schedules as S_{min} and S_{max}, see Figure 12. In particular, S_{min} is a schedule where all tasks are assumed to end at the minimum possible time and are therefore sorted by increasing $\min(end(t_i))$; conversely, in S_{max} tasks are assumed to end at the maximum possible time, hence they are ordered according to $\max(end(t_i))$. Obviously both situations will likely be infeasible, but have to be taken into account. Moreover, the following theorem holds:

Theorem 5 *The expected makespan assumes the maximum possible value in the S_{max} schedule, the minimum possible value in the S_{min} schedule.*

Proof: Let us take into account S_{max}. Let $t_0, \ldots, t_{n-1}$ be the respective task order; the corresponding expected makespan value due to expression 5 is a weighted sum of (maximum) end times:

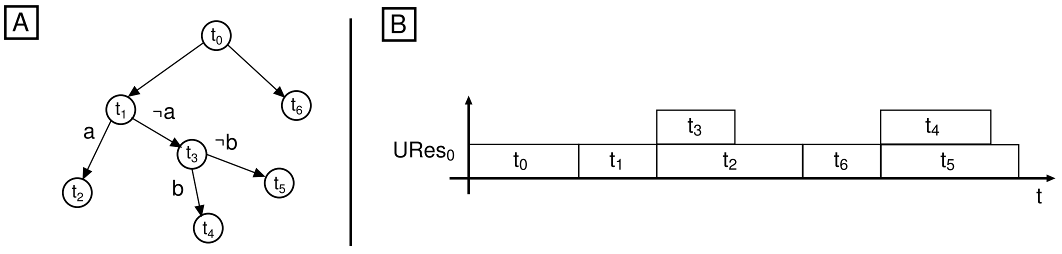


Fig. 11. Temporal task grouping

$$emkspan(S_{max}) = w_0 \cdot max(end(t_0)) + \cdots + w_{n-1} \cdot max(end(t_{n-1}))$$

Note that $\sum_i w_i = 1$ as weights are probability; also note weights are univocally defined by the task order. If S_{max} were not the expected maximum makespan schedule, it should be possible to increase the expected makespan value by reducing the end time of some tasks. Now, let us gradually decrease $max(end(t_i))$ while maintaining $max(end(t_i)) \geq max(end(t_{i-1}))$: as long as w_i does not change the expected makespan value necessarily decreases. When $max(end(t_i))$ gets lower than $max(end(t_{i-1}))$, weights w_i and w_{i-1} change as follows:

$$\begin{aligned} w_i = p(t_i \wedge \neg t_{i+1} \wedge \ldots \wedge \neg t_{n-1}) &\rightarrow p(t_i \wedge \neg t_{i-1} \wedge \neg t_{i+1} \wedge \ldots \wedge \neg t_{n-1}) \\ w_{i-1} = p(t_{i-1} \wedge \neg t_i \wedge \neg t_{i+1} \wedge \ldots \wedge \neg t_{n-1}) &\rightarrow p(t_{i-1} \wedge \neg t_{i+1} \wedge \ldots \wedge \neg t_{n-1}) \end{aligned}$$

hence w_{i-1} gets higher, w_i gets lower. As the sum $\sum_i w_i$ is constant and equal to 1 both before and after the swap, w_{i-1} grows exactly by the amount by which w_i shrinks; in other terms, some of the weight of t_i is transferred to t_{i-1} or, equivalently, t_{i-1} "steals" some weight from t_i. From now on, if we keep on decreasing $max(end(t_i))$ the expected makespan will still decrease, just at a slower pace due to the lower value w_i, until w_i will become 0. Hence by reducing the end time of a single time variable the expected makespan can only get worse. Moving more tasks complicates the situation, but the reasoning still holds. An analogous proof can be done for the expected makespan for the S_{min} schedule. □

We can therefore prune the expected makespan variable by enforcing:

$$emkspan(S_{min}) \leq emkspan \leq emkspan(S_{max}) \tag{6}$$

In order to improve computational efficiency, we can use F-nodes of the BFG instead of tasks in the computation of $emkspan(S_{min})$ and $emkspan(S_{max})$, exploiting the mapping between tasks (CTG nodes) and F-nodes; for details see Appendix B.

Pruning the makespan variable requires to compute the makespan of the two schedules S_{min}, S_{max}; this is done by performing a BFG query (complexity $O(n_t)$) and a probability computation (complexity $O(n_t)$) for each task ($O(n_t)$ iterations). The basic worst case complexity is therefore $O(n_t^2)$, which can be reduced to $O(n_t \log(n_t))$ by exploiting caching and dynamic updates during search. As an intuition, all probability weights in the BFG can be computed at the root of the search tree and cached. Then, each time a variable $end(t_i)$ changes, possibly some nodes change their positions in S_{min}, S_{max} (see Figure 12B, where $\max(end(t_1))$ changes and becomes lower than $\max(end(t_3))$,

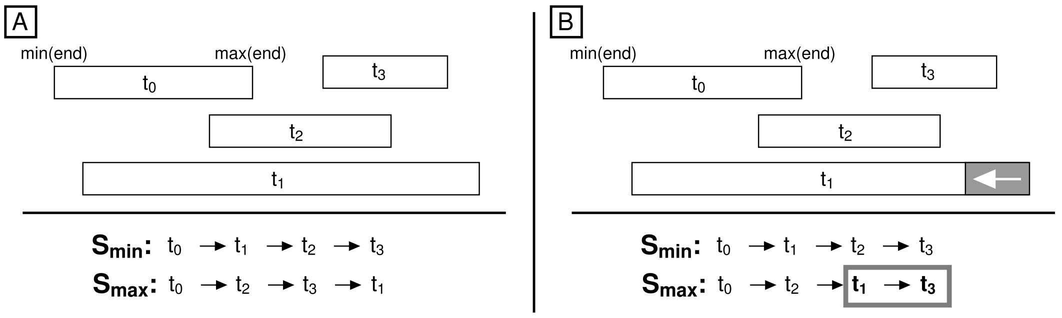


Fig. 12. A: Example of S_{min} and S_{max} sequences. B: An update of $end(t_1)$ causes a swap in S_{max}

.

thus the two nodes are swapped); in such a situation, the probabilities of all the re-positioned nodes have to be updated. Each update is done in $O(\log(n_t))$ by modifying the probability weights on the BFG; as no more than n_t nodes can move between a search node and any of its children, the overall complexity is $O(n_t \log(n_t))$.

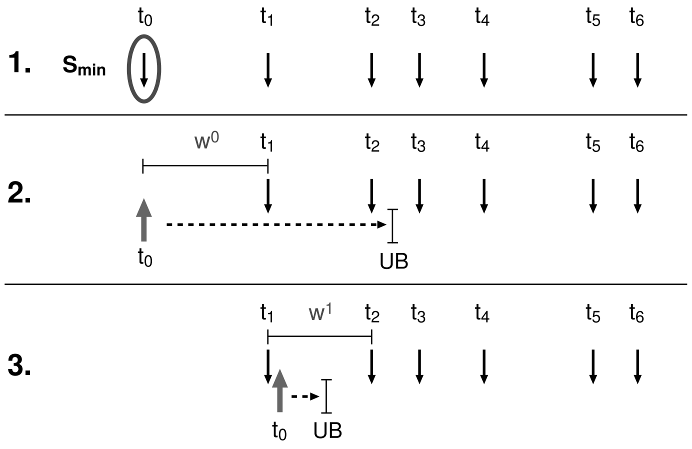


Fig. 13. Upper bound on end variables

6.2.2 *Filtering end time variables*

When dealing with a makespan minimization problem, it is crucial for the efficiency of the search process to exploit the makespan variable domain updates (e.g. when a new bound is discovered) to filter the end variables domains.

Bounds for $end(t_i)$ can be computed again with expression (5); for example to compute the upper bound for $end(t_i)$) we have to subtract from the maximum expected makespan value ($\max(mkspan)$) the minimum contribution of all tasks except t_i:

$$end(t_i) \leq \frac{\max(emkspan) - \sum_{j \neq i} p(t_j \wedge \neg t_{j+1} \wedge \ldots) \min(end(t_j))}{p(t_i \wedge \neg t_{i+1} \wedge \ldots)} \tag{7}$$

where $t_0, \dots, t_{i-1}, t_i, t_{i+1}, \dots$ is the sequence where the contribution of $t_j, j \neq i$ is minimized. Unfortunately, this sequence is affected by the value of $end(t_i)$. In principle, we should compute a bound for all possible assignments of $end(t_i)$, while keeping the contribution of other nodes minimized.

Note that the sequence where the contribution of *all* tasks is minimized is that of the S_{min} schedule; we can compute a set of bounds for $end(t_i)$ by "sweeping" its position in the sequence, and repeatedly applying formula (7). An example is shown in Figure 13, where a bound is computed for t_0 (step 1 in Figure 13). We start by computing a bound based on the current position of t_0 in the sequence (step 2 in Figure 13); if such a bound is less than $\min(end(t_1))$, then $\max(end(t_0))$ is pruned, otherwise we swap t_0 and t_1 in the sequence and update the probabilities (the original probability w^0 becomes w^1) according to expression (5). The process continues by comparing t_0 with t_2 and so on until $\max(end(t_0))$ is pruned or the end of S_{min} is reached. Lower bounds for $\min(end(t_i))$ can be computed similarly, by reasoning on S_{max}.

A detailed description of the filtering procedure is given in Algorithm 2. The tasks are processed as they appear in S_{min} (line 2); for each t_j the algorithm starts to scan the next intervals (line 6). For each interval we compute a bound (lines 7 to 11) based on the maximum makespan value ($\max(mkspan)$), the current task probability/weight (wgt) and the contribution of all other tasks to the makespan lower bound ($rest$).

If the end of the list is reached or the bound is within the interval (line 12) we prune the end variable of the current task (line 13) and the next task is processed. If the bound exceeds the current interval, we move to the next one. In the transition the current task possibly gains weight by "stealing" it from the activity just crossed (lines 15 to 18); wgt and $rest$ are updated accordingly.

The algorithm takes into account all tasks (complexity $O(n_t)$) and for each of them it analyzes the subsequent intervals (complexity $O(n_t)$); weights are updated at each transition with complexity $O(\log(n_t))$ taking care of the fact that a task can be mapped to more than one F-node (note that directly working on F nodes avoids this issue). The overall complexity is $O(n_t^2 \log(n_t))$; by manipulating F-nodes instead of tasks it can be reduced down to $O(n_t + n_o^2 \log(n_o))$, where n_o is the number of condition outcomes in the CTG, see Appendix B.

7 CONDITIONAL CONSTRAINTS

To tackle scheduling problems of conditional task graphs we introduced the so called *conditional constraints*, which extend traditional constraints to take into account the feasibility in all scenarios.

Algorithm 2 End variables pruning (upper bound)

```
1: let $S_{min} = t_0, t_1, \ldots, t_{k-1}$
2: for $j = 0$ to $k-1$ do
3:   compute result of query $q = t_j \wedge \neg t_{j+1} \wedge \ldots \wedge t_{k-1}$ and probability $p(q)$
4:   $wgt = p(q)$
5:   $rest = mkspan(S_{min}) - \min(end(t_j))wgt$
6:   for $h = j$ to $k-1$ do
7:     if $wgt > 0$ then
8:       $UB = \frac{\max(mkspan) - rest}{wgt}$
9:     else
10:      $UB = \infty$
11:    end if
12:    if $h = (k-1)$ or $UB \leq \min(end(t_{h+1}))$ then
13:      set $UB$ as upper bound for $t_j$
14:    else
15:      remove element $\neg t_{h+1}$ from query $q$ and update $p(q)$
16:      $newwgt = p(q)$
17:      $rest = rest - (newwgt - wgt)\min(end(t_{h+1}))$
18:      $wgt = newwgt$
19:    end if
20:  end for
21: end for
```

Let C be a constraint defined on a set of variables X, let S be the set of scenarios of a given CTG, let $X(s) \subseteq X$ be the set of variables related to tasks appearing in the scenario $s \in S$. The conditional constraint corresponding to C must enforce:

$$\forall s \in S \quad C|_{X(s)}$$

where $C|_{X(s)}$ denotes the restriction of constraint C to variables in $X(s)$.

A very simple example is the disjunctive conditional constraint [18] that models temporal relations between tasks t_i and t_j that need the same unary resource for execution. The disjunctive constraint enforces:

$mutex(t_i, t_j) \vee (end(t_i) \leq start(t_j)) \vee (end(t_j) \leq start(t_i))$

where *mutex(t_i,t_j)* holds if tasks t_i and t_j are mutually exclusive (see Definition 6) so that they can access the same resource without competition.

As another example, let us consider the cumulative constraint modeling limited capacity resources. The traditional resource constraint enforces for each time instant t:

$$\forall \text{ time } t, \forall R \in \bigcup_{t_i} ResSet_i \sum_{\substack{t_i : \\ start(t_i) \leq t < end(t_i) \\ res(t_i) = R}} ResCons_i \leq Cap(R)$$

while its conditional version enforces:

$$\forall \text{ time } t, \forall s \in S, \, \forall R \in \bigcup_{t_i \in TG(s)} ResSet_i \sum_{\substack{t_i \in TG(s) : \\ start(t_i) \leq t < end(t_i) \\ res(t_i) = R}} ResCons_i \leq Cap(R)$$

where the same constraint as above must hold *for every scenario*; this indeed amounts to a relaxation of the deterministic case.

As a consequence, resource requirements of mutually exclusive tasks are not summed, since they never appear in the same scenario. In principle, a conditional constraint can be implemented by checking the correspondent non conditional constraint for each scenario; however, the number of scenarios in a CTG grows exponentially with the number of branch nodes and a case by case check is not affordable in practice. Therefore, implementing conditional constraints requires an efficient tool to reason on CTG scenarios; this is provided by the BFG framework, described in Section 5.1.

We have defined and implemented the conditional version of the timetable constraint [23] for cumulative resources described in the following section; other conditional constraints can be implemented by using the BFG framework and taking inspiration from existing filtering algorithms.

7.1 Timetable constraint

A family of filtering algorithms for cumulative resource constraints is based on timetables, data structures storing the worst case usage profile of a resource over time [23]. While timetables for traditional resources are relatively simple and very efficient, computing the worst usage profile in presence of alternative scenarios and mutually exclusive activities is not trivial, since it varies in a non linear way; furthermore, every activity has its own resource view.

Suppose for instance we have the CTG in Figure 14A; tasks t_0, ..., t_4 and t_6 have already been scheduled: their start time and durations are reported in Figure 14B; all tasks require a single cumulative resource of capacity 3, and the requirements are reported next to each node in the graph. Tasks t_5 and t_7 have not been scheduled yet; t_5 is present only in scenario $\neg a$, where the resource usage profile is the first one reported in Figure 14B; on the other hand, t_7 is present only in scenario a, b, where the usage profile is the latter in Figure 14B. Therefore the resource view at a given time depends on the activity we are considering. In case an activity is present in more than one scenario, the worst case at each time instant has to be considered.

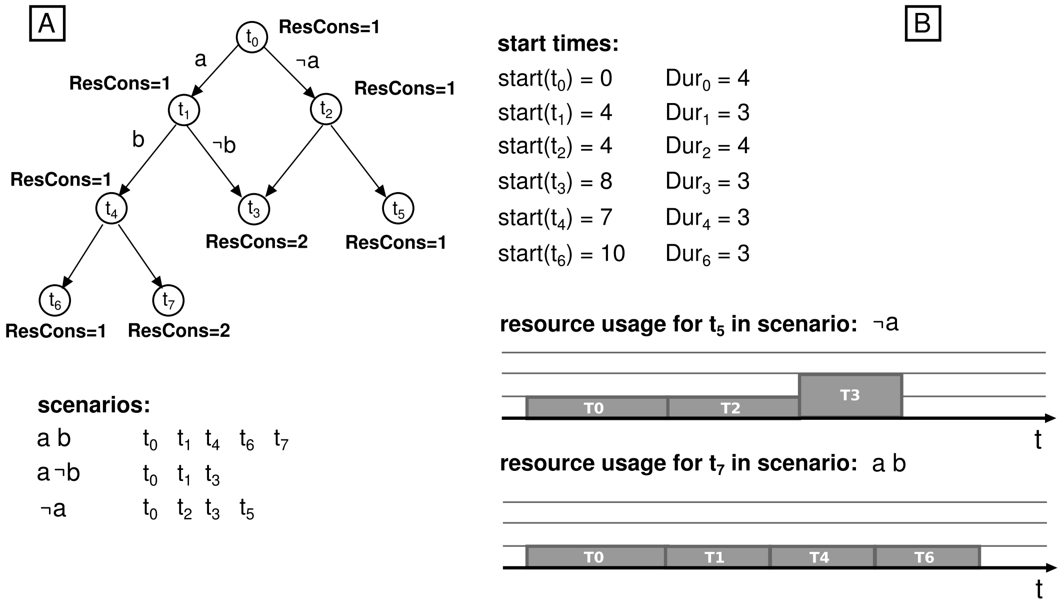


Fig. 14. Capacity of a cumulative resource on a CTG

We introduce a new global timetable constraint for cumulative resources and conditional tasks in the non-preemptive case. The global constraint keeps a list of all known starting and ending points of activities (in particular their latest start times and earliest end times); given an activity t_i, if $lst(t_i) \leq eet(t_i)$ then the activity has an *obligatory part* from $lst(t_i)$ to $eet(t_i)$ contributing to the resource profile.

The filtering algorithm is described in Algorithm 3. All along the algorithm t_i is the target activity, the variable "*time*" represents the time point currently under exam and "$finish$" is finish line value (when it is reached the filtering is over); finally "$firstPStart$" represents the first time point where t_i can start and"*good*" is a flag whose value is false if the resource capacity is exceeded at the last examined time point.

Algorithm 3 keeps on scanning meaningful end points of all obligatory parts in the interval $[est(t_i), finish)$ until (line 4):

(1) The resource capacity is exceeded in the current time point ($good = false$) and the current time point has gone beyond the latest start time of t_i (in this case the constraint fails)

(2) The resource capacity is not exceeded in the current time point ($good = true$) and the finish line has been reached ($time \geq finish$)

Next, the resource usage is checked at the current time point (line 5); in case the capacity is exceeded this is recorded ($good = false$ at line 7) and the algorithm moves to the next *end* point of an obligatory part ($eet(t_j)$) in the hope the resource will be freed by that time. In case the capacity is not exceeded: (A) the current time point becomes suitable for the activity to start (line 10) and (B) the finish line is updated (line 11) to the current time value plus the duration of the activity; then the algorithm keeps on checking the starting time of obligatory parts (see line 14). If the finish line is reached without reporting a resource over-usage, then the start time of t_i can be updated (line 18).

Algorithm 3 Filtering algorithm for the conditional timetable constraint

```
 1: let $time = est(t_i)$, $finish = eet(t_i)$
 2: let $firstPStart = time$
 3: let $good = true$
 4: while  $\neg [(good = false \wedge time > lst(t_i)) \vee (good = true \wedge time >= finish)]$
    do
 5:   if $ResCons_i + resUsage(t_i, time) > resCapacity$ then
 6:     let $time =$ next $eet(t_j)$
 7:     let $good = false$
 8:   else
 9:     if $good = false$ then
10:       let $firstPStart = time$
11:       let $finish = max(finish, time + Dur_i)$
12:       let $good = true$
13:     end if
14:     let $time =$ next $lst(t_j)$
15:   end if
16: end while
17: if $good = true$ then
18:   let $est(t_i) = firstPStart$
19: else
20:   fail
21: end if
```

Algorithm 3 treats the computation of the resource usage as a black box: the $resUsage(t_i, time)$ denotes the worst case usage at time *time*, as seen by task t_i; the worst case usage of a cumulative resource as seen by the current activity can be computed efficiently via a backward visit, as described in Algorithm 1, on a BFG whose F-nodes are labeled with a weight value as follows.

To compute the worst case usage of a resource at time t we first have to "load" the requirement of each task t_i executing at time t (such that $lst(t_i) \leq t \leq eet(t_i)$) on each F-node F_j such that t_i belongs to the node inclusion label

$(t_i \in i(F_j))$. For the computation of the maximum weight of a scenario each F and B-node has a single attribute w representing a weight value (in particular $A = [0, \infty)$). The *init* and *update* functions are defined as follows:

$$
\begin{aligned}
& init_F(F_i) = \sum_{\substack{lst(t_j) \leq t \leq eet(t_j), \\ t_j \in i(F_i)}} ResCons_i \\
& init_B(B_i) = 0 \\
& update_F(F_i, B_j) = \max(w(B_j), w(F_i)) \\
& update_B(B_i, F_j) = w(F_j) + w(B_i)
\end{aligned}
$$

At the end of the process the weight of the root node is the worst case resource usage.

Basically Algorithm 1, instantiated as described, performs a backward visit of the BFG, summing up the weight of each child for every F-node (see $update_B$) and choosing for each B-node the maximum weight among those of its children (see $update_F$). As each outcome has to be processed once, the complexity is $O(n_o)$; loading a CTG task on an F-node has complexity $O(n_o)$. The timetable filtering algorithm Algorithm 3 in the worst case loads a CTG node and computes a weight for each task, hence n_t times; therefore the complexity of the filtering algorithm for the time window of a single task is $O(n_t(n_o + n_o)) = O(n_t n_o)$. This value can be reduced by caching the results and updating the data structures when a time window (say of task t_i) is modified; this is done by updating data on F-nodes and propagating the change backward along the BFG; due to its tree like structure this is done in $O(\log(n_o))$ and the overall complexity is reduced to $O(n_t \log(n_o))$

8 Related work

This paper is a substantially revised and extended version of two previous papers: in [25] we propose a similar framework for dealing with objective functions depending on task allocation in the field of embedded system design, while in [24] we face the makespan minimization problem. In the present paper, we recall some ideas of these previous papers, but in addition we describe conditional constraints, we formalize the overall stochastic framework and we perform extensive evaluation.

The area where CTG allocation and scheduling has received more attention is most probably the one of embedded system design. In this context, Conditional Task Graphs represent a functional abstraction of embedded applications that should be optimally mapped onto multi core architectures (Multi Processor Systems on Chip - MPSoCs). The optimal allocation and schedule guarantees high performances for the entire life time of the system. The problem has been faced mainly with incomplete approaches: in particular, [12] is one of the earliest works were CTGs are referred to as Conditional Process Graphs; there the focus is on minimizing the worst case completion time and a solution is provided by means of a branch outcome dependent “schedule table”; a list scheduling based heuristic is provided and inter tasks communications are taken into account as well. In [42] a genetic algorithm is devised on the basis of a conditional scheduling table whose (exponential number of) columns represent the combination of conditions in the CTG and whose rows are the starting times of activities that appear in the scenario. The size of such a table can indeed be reasonable in real world applications. Another incomplete approach is described in [39] that proposes a heuristic algorithm for task allocation and scheduling based on the computation of mutual exclusion between tasks. Finally, [31] describes an incomplete algorithm for minimizing the energy consumption based on task ordering and task stretching.

To our knowledge, beside our previous work on CTG, the only complete approach to the CTG allocation and scheduling problem is proposed in [18] and is based on Constraint Programming. The solving algorithm used only scales up to small task graphs ($\sim$ 10 activities) and cumulative resources are not taken into account. Only a simple unary resource constraint is implemented in the paper. Also, the expected value of the objective function is not taken into account.

Another complete CP based approach is described in [19] and targets low level instruction scheduling with Hierarchical Conditional Dependency Graphs (HCDG); conditional dependencies are modeled in HCDGs by introducing special nodes (guards), to condition the execution of each operation; complexity blowup is avoided by providing a single schedule where operations with mutually exclusive guards are allowed to execute at the same time step even if they access the same resource. We basically adopted the same approach to avoid scheduling each scenario independently. Mutual exclusions relations are listed in HCDGSs for each pair of tasks and are computed off line by checking compatibility of guard expressions, whereas in CTGs they are deduced from the graph structure; note the in-search computation described in the paper is just used to support speculative execution. Pairwise listing of exclusion relations is a more general approach, but lacks some nice properties which are necessary to efficiently handle non-unary capacity resources; in particular computing worst case usage of such a resource is a NP-complete problem if only pairwise mutual exclusions are known; in fact, in [19] only unary resources are taken into account.

An interesting framework where CTG allocation and scheduling can fit is the one presented in [9]. The framework is general taking into account also other forms of stochastic variables (say for example task durations) and can integrate three general families of techniques to cope with uncertainty: proactive techniques that use information about uncertainty to generate and solve a problem model; revision techniques that change decisions when it is relevant during execution and progressive techniques that solve the problem piece by piece on a gliding time horizon. Our paper is less general and more focused on the efficient solution of a specific aspect of the framework, namely conditional branches and alternative activities.

Conditional Task Graph may arise in the context of conditional planning [28]. Basically, in conditional planning we have to check that each execution path (what we call scenario) is consistent with temporal constraints. For this purpose extensions to the traditional temporal constraint based reasoning have been proposed in [36] and [35]. However, these approaches do not take into account the presence of resources, which is conversely crucial in constraint based scheduling

Other graph structures similar to CTG have been considered in [20], [7]. These graphs contain the so called *optional activities* but their choice during execution is decided by the scheduler and is not based on the condition outcome. Basically, constraint based scheduling techniques should be extended to cope with these graphs, but no probability reasoning is required. For graphs with optional activities, an efficient unary resource constraint filtering algorithm is proposed in [38].

Close in spirit to optional activities are Temporal Networks with Alternatives (TNA), introduced in [5]. TNA augment Simple Temporal Networks with alternative subgraphs, consisting of a “principal node” and several arcs to the same number of “branching nodes”, from which the user has to choose one for run-time execution. Like in optional activities, the user is responsible for choosing a node; unlike in optional activities, exactly one branching node has to be selected. TNA do not allow early exits (as our approach does) and do not require any condition such as CFU. The follow-up work [6] proposes a heuristic algorithm to identify equivalent nodes in a TNA, similarly to what we do with the BFG. Note however that F-nodes do not necessarily represent equivalence classes (think of the fact that a CTG node can be mapped to more than one F-node), but rather elementary groups of scenarios where a subset of tasks execute.

Speaking more generally, stochastic problems have been widely investigated both in the Operations Research community and in the Artificial Intelligence community.

Operations Research has extensively studied stochastic optimization. The main approaches can be grouped under three categories: sampling [2] consisting of approximating the expected value with its average value over a given sample; the *l-shaped* method [22] which faces stochastic problems with recourse, i.e. featuring a second stage of decision variables, which can be fixed once the stochastic variables become known. The method is based on Benders Decomposition [8]; the master problem is a deterministic problem for computing the first phase decision variables. The subproblem is a stochastic problem that assigns the second stage decision variables minimizing the average value of the objective function. A third method is based on branch and bound extended for dealing with stochastic variables, [26].

In the field of stochastic optimization an important role is played by stochastic scheduling. This field is motivated by problems arising in systems where scarce resources must be allocated over time to activities with random features. The aim of stochastic scheduling problems is to come up with a policy that, say, prioritize over time activities awaiting service. Mainly three methodologies have been developed:

- Models for scheduling a batch of stochastic jobs, where the tasks to be scheduled are known but their processing time is random, with a known distribution, (see the seminal papers [32,29]).
- Multi armed bandit models [14] that are concerned with the problem of optimally allocating effort over time to a collection of projects which change state in a random fashion.
- Queuing scheduling control models [10] that are concerned with the design of optimal service discipline, where the set of activities to be executed is not known in advance but arrives in a random fashion with a known distribution.

Temporal uncertainty has also been considered in the Artificial Intelligence community by extending Temporal Constraint Networks (TCSP) to allow contingent constraints [37] linking activities whose effective duration cannot be decided by the system but is provided by the external world. In these cases, the notion of consistency must be redefined in terms of controllability; intuitively, a network is controllable if it is consistent in any situation (i.e. any assignment of the whole set of contingent intervals) that may arise in the external world. Three levels of controllability must be distinguished, namely the strong, the weak and the dynamic one. We ensure in this paper strong controllability since we enforce consistency in all scenarios.

The Constraint Programming community has recently faced stochastic problems: in [40] stochastic constraint programming is formally introduced and the concept of solution is replaced with the one of *policy*. In the same paper, two algorithms have been proposed based on backtrack search. This work has been extended in [33] where an algorithm based on the concept of scenarios is proposed. In particular, the paper shows how to reduce the number of scenarios and still provide a reasonable approximation of the value of optimal solution. We will compare our approach to the one reported in this paper both in terms of efficiency and solution quality.

9 Experimental results

Our approach, referred to as conditional solver, has been implemented using the state of the art ILOG Cplex 11.0, Solver 6.3 and Scheduler 6.3. We tested the approach on a number of instances representing several variants of a real world hardware design problem, where a multi task application (described by means of a CTG) has to be scheduled on a multiprocessor hardware platform. The problem features complex precedence relations (representing data communications), unary resources (the processors) and a single cumulative resource (modelling a shared communication channel whose capacity is its total bandwidth).

Instances for all problem variants are "realistic", meaning that they are randomly generated on the base of real world instances [15]. We designed groups of experiments for two variants of the problem (described respectively in Sections 9.1 and 9.2), to evaluate the conditional timetable constraint and objective functions presented in this paper and the performance of the BFG framework. Also, we compare our approach with a scenario based solver [33] that explicitly considers all scenarios or a subset of them.

9.1 Bus traffic minimization problem

In the first problem variant hardware resources like processing elements and memory devices have to be allocated to tasks in order to minimize the expected bus traffic. Once all resources are assigned, tasks have to be scheduled and a specified global deadline must be met. The objective depends only on the allocation choices and counts two contributions: one depending on single task-resource assignments, one depending on pairs of task-resource assignments. Basically, the objective function captures both cases described in Section 6.1.

We faced the problem by means of Logic Based Benders' Decomposition [16] as explained in [25], where the master problem is the resource allocation and the subproblem is the computation of a feasible schedule.

We implemented a conditional solver based on BFG and a scenario based one [33]. In the first case the stochastic objective function in the master problem is reduced to a deterministic expression where scenario probabilities are computed as described in Section 6.1; in the scheduling subproblem unary resources (the processors) are modeled with conditional disjunctive constraints, while the communication channel is considered a cumulative resource and is modeled with a conditional timetable constraint.

In the scenario based solver the objective function is a sum of an exponential number of linear terms, one for each scenario. Processors are modeled again with conditional disjunctive constraints, while for the communication channel a collection of discrete resources (one per scenario) is used. A simple scenario reduction technique is implemented, so that the solver can be configured to take into account only a portion of the most likely scenarios.

We generated 200 instances for this problem, ranging from 10 to 29 tasks, 8 to 33 arcs, which amounts to 26 to 95 activities in the scheduling subproblem (tasks and arcs are split into several activities). All instances satisfy Control Flow Uniqueness, and the number of scenario ranges from 1 to 72. The CTG generation process works by first building a deterministic Task Graph, and then randomly selecting some fork nodes to be turned into branches (provided CFU remains satisfied); outcome probabilities are chosen randomly according to a uniform distribution. The number of processors in the platform goes from 2 to 5. We ran experiments on a Pentium IV 2GHz with 512MB of RAM with a time limit of 900 seconds.

						time			
tasks	**arcs**	**acts**	**scens**	**proc**	**min**	**med**	**max**	$>$ **TL**	**inf**
10-12	8-12	26-36	1-6	2-2	0.02	0.07	25.41	0	3
12-14	10-16	32-46	3-9	2-2	0.04	0.12	610.96	1	1
14-15	12-17	38-49	2-9	2-3	0.03	0.22	33.70	0	7
15-18	13-22	41-62	2-18	3-3	0.11	0.43	2.56	1	7
18-19	16-22	50-63	4-30	3-3	0.31	1.78	87.52	2	2
20-21	16-25	52-71	3-24	4-4	0.29	1.68	741.49	2	4
21-23	19-28	59-79	6-24	4-4	1.27	1.27	641.74	3	11
24-25	20-29	64-83	4-36	4-5	0.73	3.25	479.16	2	5
25-28	22-30	69-88	5-72	5-5	0.19	2.37	382.36	4	7
28-29	23-33	74-95	8-48	5-5	2.49	11.49	78.46	4	4

Table 1
Performance tests for the expected bus traffic minimization problem

The first set of experiments, reported in Table 1, has the goal to test the performance of the conditional solver. Each row refers to a group of 20 instances and reports the minimum and maximum number of tasks, arcs, scheduling activities scenarios and processors (columns: tasks, arcs, acts, scens and proc); minimum (column: min), median (med) and maximum (max) computation time for the instances solved to optimality, included the time to perform preprocessing and build the BFG. Then the number of instances not solved within the time limit follows ($>$ TL) and the number of infeasible instances (inf). As one can see the median computation time is pretty low and grows with the size of the instance, while its maximum has a more erratic behavior, influenced by the presence of uncommonly difficult instances. The number of timed-out instances intuitively grows with the size of the graph.

		S100			S80				S50			
scens	**proc**	$\frac{\mathbf{T}}{\mathbf{T_{cond}}}$	$>$ **TL**	**inf**	$\frac{\mathbf{T}}{\mathbf{T_{cond}}}$	$>$ **TL**	**inf**	$\frac{\mathbf{Z}}{\mathbf{Z_{cond}}}$	$\frac{\mathbf{T}}{\mathbf{T_{cond}}}$	$>$ **TL**	**inf**	$\frac{\mathbf{Z}}{\mathbf{Z_{cond}}}$
1-3	2-3	1.30	1	3	1.31	1	3	1.00	0.85	1	3	0.86
3-4	2-4	0.98	2	5	1.22	2	5	1.00	0.57	1	1	0.74
4-6	2-5	1.33	2	4	0.85	1	4	0.97	1.04	0	4	0.69
6-6	2-5	1.06	0	5	1.31	0	5	0.96	0.83	0	4	0.78
6-8	2-4	0.96	1	6	1.15	1	6	0.93	1.20	0	6	0.69
8-9	2-5	0.89	6	5	1.00	6	5	0.88	0.48	5	4	0.62
9-12	2-5	1.24	3	6	1.11	4	6	0.96	0.79	3	6	0.79
12-12	3-5	1.38	3	8	0.96	4	8	0.91	0.97	4	8	0.73
12-24	3-5	0.98	2	6	0.96	3	6	0.87	0.59	3	5	0.79
24-72	3-5	1.21	3	2	1.04	3	2	0.81	0.89	3	2	0.64

Table 2
Performance tests for the expected bus traffic minimization problem

Then we compared the conditional solver we realized with a scenario based one for the same problem: the results for this second group of tests are shown in Table 2. Each row reports results for a group of 20 instances, for which it shows the minimum and maximum number of scenarios (column scens), the minimum and maximum number of processors (procs), and some data about the scenario based solver when 100% (S_{100}) 80% (S_{80}) and 50% (S_{50}) of the most likely scenarios are considered. In particular we report for each scenario based solver the average solution time ratio with the conditional solver ($\frac{T}{T_{cond}}$, on the instances solved by both approaches), the number of timed out instances ($>$ TL, not considered in the average time computation) and the number of infeasible instances (inf). For the S50 and S80 solvers also the average solution quality ratio is shown ($\frac{Z}{Z_{cond}}$). Note that in Table 2, instances are sorted by number of scenarios, rather than by size; as a consequence, the first rows do not necessarily refer to the smallest nor the easiest scheduling problems.

On average, the conditional solver improves the scenario based one by a 13% factor; this is not an impressive improvement. Also the improvement does not occur in all cases; the reason is that the computation time for this problem is dominated by that of finding an optimal resource allocation, and with regard to this subproblem the conditional approach only offers a more efficient way to build the *same* objective function expression. We expect to have much better results as the importance of the scheduling subproblem grows.

Note how the use of scenario reduction techniques speeds up the solution process for S_{80} and S_{50}, but introduces inaccuracies in the objective function value, which is lower than it would be (see column $\frac{Z}{Z_{cond}}$). Also, some infeasible instances are missed (the value of the "inf" column for S_{50} is lower than S_{100}).

In the second problem variant we consider the minimization of the expected makespan, that is the expected application completion time. This is indeed much more complex than the previous case, since this objective function depends on the scheduling related variables. We therefore chose to limit ourselves to computing an optimal schedule for a given resource allocation.

As we did for the previous problem variant, we implemented both a conditional and a scenario based solver. In the conditional solver the makespan computation is handled by the global constraint described in Section 6.2, whereas in the scenario based solver the makespan is the sum of the completion time of each possible scenario, weighted by the scenario probability (see expression 4). Processors and bus constraints are modeled as described in Section 9.1.

For this problem we generated 800 instances, ranging from 37 to 109 tasks, 2 to 5 “heads” (tasks with no predecessor), 3 to 11 “tails” (tasks with no successor), 1 to 135 scenarios. The number of processors (unary resources) ranges from 3 to 6. Again all instances satisfy the control flow uniqueness. We ran experiments with a time limit of 300 seconds; all tests were executed on a AMD Turion 64, 1.86 GHz.

We performed a first group of tests to evaluate the efficiency of the expected makespan conditional constraint and the quality of the solutions provided(in particular the amount of gain which can be achieved by minimizing the expected makespan compared to worst case based approaches); a second group of experiment was then performed to compare the performances of the conditional solver with the scenario-based one. Table 3 shows the results for the first group of tests; here we evaluate the performance of the solver using the conditional timetable constraint (referred to as C) and compare the quality of the computed schedules versus an identical model where the deterministic makespan is minimized (referred to as W). In this last case, no expected makespan constraint is used; the objective function is thus deterministic and amounts to minimizing the worst case makespan (hence the objective for the deterministic model will be necessarily worse than the conditional one). The models for C and W are identical with every other regard (they both use conditional resource constraints and assign a fixed start time to every task). Each row identifies a group of 50 instances. For each group we report the minimum and maximum number of activities (acts), of scenarios (scens) and of unary resources (proc), the average solution time (T(C)), the average number of fails (F(C)) and the number of instances which could not be solved within the time limit ($>$ TL) by the conditional solver.

In column C/W we report the makespan value ratio which shows an average improvement of 12% over the deterministic objective. The gain is around 16% if we consider only the instances where the makespan is actually improved (column stc C/W). The computing time of the two approaches is surprisingly roughly equivalent for all instances.

acts	**scens**	**proc**	**T(C)**	**F(C)**	**> TL**	$\frac{\mathbf{C}}{\mathbf{W}}$	**stc** $\frac{\mathbf{C}}{\mathbf{W}}$
37-45	1-2	3-4	1.54	3115	0	0.83	0.80
45-50	1-3	3-5	2.67	4943	0	0.88	0.84
50-54	1-3	3-5	9.00	17505	0	0.88	0.85
54-57	2-4	4-5	25.68	52949	1	0.88	0.85
57-60	1-6	4-5	29.78	77302	1	0.94	0.90
60-65	1-6	4-6	24.03	28514	0	0.85	0.80
65-69	2-8	4-6	32.12	47123	2	0.90	0.84
69-76	3-12	4-6	96.45	101800	14	0.86	0.82
76-81	1-20	5-6	144.67	134235	21	0.90	0.86
81-86	3-24	5-6	143.31	130561	17	0.84	0.75
86-93	2-36	5-6	165.74	119930	25	0.93	0.87
93-109	4-135	5-6	185.56	127321	28	0.93	0.87

Table 3
Performance tests

Table 4 compares the conditional model with a scenario based solver; we remind that in this second case the cumulative resource is implemented with one constraint per scenario and the expected makespan is expressed with the declarative formula (4). In both models unary resources (processors) are implemented with conditional constraints.

Again, rows of Table 4 report average results for groups of 50 instances; instances are grouped and sorted by increasing number of scenarios; hence, once again the results on the first row do not necessarily refer to the easiest/smallest instances. The table reports the solution time of the conditional solver (T(C))

scens	**T(C)**	**> TL**	$\frac{\mathbf{T(S_{100})}}{\mathbf{T(C)}}$	**> TL**	$\frac{\mathbf{T(S_{80})}}{\mathbf{T(C)}}$	**> TL**	$\frac{\mathbf{S_{80}}}{\mathbf{C}}$	$\frac{\mathbf{T(S_{50})}}{\mathbf{T(C)}}$	**> TL**	$\frac{\mathbf{S_{50}}}{\mathbf{C}}$
1-2	41.00	5	22.60	5	22.66	5	1.00	0.58	3	0.77
2-3	66.02	8	19.85	10	19.93	10	1.00	1.69	7	0.80
3-4	43.80	5	35.05	8	35.12	8	1.00	9.19	5	0.79
4-5	49.94	6	73.75	9	73.63	9	1.00	57.03	8	0.80
5-6	66.39	9	48.74	12	16.64	12	0.98	0.77	8	0.82
6-6	51.26	5	6.52	8	6.11	8	0.96	41.99	8	0.80
6-8	38.85	5	82.21	11	71.09	9	0.98	84.41	3	0.80
8-9	57.78	9	66.32	10	63.70	10	0.98	26.76	9	0.85
9-12	52.96	5	89.52	13	86.97	13	0.98	40.43	6	0.85
12-14	117.93	17	45.60	22	43.02	22	0.97	37.35	18	0.84
14-20	95.74	11	32.62	22	31.85	21	0.99	28.76	15	0.90
20-135	178.88	24	66.19	37	65.56	37	1.00	22.09	35	0.912

Table 4
Comparison with scenario based solver

and the performance ratios w.r.t the scenario based solver with 100% (S_{100}), 80% (S_{80}) and 50% (S_{50}) of the most likely scenarios. The four columns "> TL" show the number of instances not solved within the time limit for each approach. Finally, columns S_{50}/C and S_{80}/C show the accuracy of the solution provided by S_{50} and S_{80} solvers.

As it can be seen the conditional model now outperforms the scenario based one by an average factor of 49.08. For this problem, in fact, the conditional approach provides a completely different and more efficient representation of the objective function, rather then just a more efficient procedure to build the same expression (as it was the case for the traffic minimization).

By reducing the number of considered scenarios the performance gap decreases; nevertheless, the conditional solver remains always better than S_{80}; it is outperformed by S_{50} when the number of scenarios is low, but the solution provided has an average 17% inaccuracy. Moreover, neither S_{50} nor S_{80} guarantee feasibility in all cases, since some scenarios are not considered at all in the solution.

10 Conclusion

CTG allocation and scheduling is a problem arising in many application areas that deserves a specific methodology for its efficient solution. We propose to use a data structure, called Branch/Fork graph, enabling efficient probabilistic reasoning. BFG and related algorithms can be used for extending traditional constraint to the conditional case and for the computation of the expected value of a given objective function.

The experimental results show that the conditional solver is effective in practice, and that it outperforms a scenario based solver for the same problem. The performance gap becomes significant when the makespan objective function is considered.

Current research is devoted to taking into account other problems where the stochastic variables are task durations or resource availability. Also, the application of CTG allocation and scheduling to the time prediction for business process management is a subject of our current research activity.

References

[1] W.M.P. van der Aalst, M.H. Schonenberg, and M. Song. Time Prediction Based on Process Mining. BPM Center Report BPM-09-04, BPMcenter.org, (2009).

http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports.htm

[2] S. Ahmed, A. Shapiro: The sample average approximation method for stochastic programs with integer recourse. Optimization on line, (2002).

[3] P. Baptiste, C. Le Pape: Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems. Constraints, 5(1/2):119-139, (2000).

[4] P. Baptiste, C Le Pape, W. Nuijten: Constraint-Based Scheduling. Kluwer Academic Publisher, (2003).

[5] R. Bartak, Ondrej Cepek: Temporal Networks with Alternatives: Complexity and Model. FLAIRS 2007: 641-646, (2007).

[6] R. Bartak, O. Cepek, P. Surynek: Modelling Alternatives in Temporal Networks IEEE Symposium on Computational Intelligence in Scheduling, SCIS07: 129-136, (2007).

[7] J.C. Beck, M. S. Fox: Constraint-directed techniques for scheduling alternative activities. Journal of Artificial Intelligence (AI) 121(1/2):211-250 (2000).

[8] J.F. Benders: Partitioning procedures for solving mixed-variables programming problems. Numerische Mathematik, 4:238-252, (1962).

[9] J. Bidot, T. Vidal, P. Laborie, J. C. Beck: A General Framework for Scheduling in a Stochastic Environment. Proc. of the Int.l Joint Conference on Artificial Intelligence, IJCAI 2007, 56-61, (2007).

[10] D.R. Cox, W.L. Smith: Queues. Chapman & Hall, (1961).

[11] R. Dechter: Constraint Processing. Morgan Kaufmann, (2003).

[12] P. Eles, Z. Peng: Bus access optimization for distributed embedded systems based on schedulability analysis. International Conference on Design and Automation in Europe, DATE2000, IEEE Computer Sociery, (2000).

[13] P. Faraboschi, J.A. Fisher, C. Young: Instruction scheduling for instruction level parallel processors. Proc. of the IEEE , 89(11):1638-1659, (2001).

[14] J.C. Gittins, D.M. Jones: A dynamic allocation index for the sequential design of experiments. Progress of Statistics, Colloq. Math Soc. Janos Bolyai, 9:241-255, (1974).

[15] A. Guerri, M. Lombardi, and M. Milano: Challenging Scheduling Problem in the field of System Design. ICAPS 2007, Workshop on Scheduling a Scheduling Competition,
(2007). available at "http://www.lia.deis.unibo.it/Staff/MicheleLombardi/", in the "Task Graph Generator" section.

[16] J.N. Hooker, G. Ottosson: Logic-based benders decomposition. Mathematical Programming, 96:33-60, (2003).

[17] K. Kennedy, R. Allen: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann, (2001).

[18] K. Kuchcinski: Constraints-driven scheduling and resource assignment. ACM Transactions on Design Automation of Electronic Systems, 8, (2003).

[19] K. Kuchcinski, C. Wolinski. Global Approach to Assignment and Scheduling of Complex Behaviors based on HCDG and Constraint Programming. Journal of Systems Architecture, 49 (12-15):489-503, (2003).

[20] J. Kuster, D. Jannach, G. Friedrich: Handling Alternative Activities in Resource Constrained Project Scheduling Problems. Proc. of the Int.l Joint Conference on Artificial Intelligence, 1960-1965, (2007).

[21] P. Laborie: Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. Journal of Artificial Intelligence, 143:151-188, (2003).

[22] G. Laporte, F.V. Louveaux: The integer l-shaped method for stochastic integer programs with complete recourse. Operations Research Letters, 13, (1993).

[23] C. Le Pape: Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-Based Scheduling Systems. Intelligent Systems Engineering, 3(2):55-66, (1994).

[24] M. Lombardi, M. Milano: Scheduling Conditional Task Graphs. Proc. of CP 2007, 468-482, (2007).

[25] M. Lombardi, M. Milano: Stochastic Allocation and Scheduling for Conditional Task Graphs in MPSoCs. Proc. of CP 2006, 299-313, (2006).

[26] V.I. Norkin, G. Pflug, A. Ruszczynski: A branch and bound method for stochastic global optimization. Mathematical Programming, 83, (1998).

[27] Organization for the Advancement of Structured Information Standards (OASIS): Web Services Business Process Execution Language Version 2.0, OASIS Standard, (2007). *http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.*

[28] M. A. Peot and D. E. Smith: Conditional nonlinear Planning. Proc. of Int.l Conference on AI Planning and Scheduling, 189-197, (1992).

[29] M.H. RothKopf: Scheduling with random service times, Management Science 12:707-713, (1966).

[30] N. Russell, W. M. P. van der Aalst, A. H. M. ter Hofstede, D. Edmond: Workflow Data Patterns: Identification, Representation and Tool Support. Proc of the 17th Int. Conference on Advanced Information Systems Engineering, 216-232, (2005).

[31] D. Shin, J. Kim: Power-aware scheduling of conditional task graphs in real-time multiprocessor systems. International Symposium on Low Power Electronics and Design (ISLPED), 408-413, ACM, (2003).

[32] W.E. Smith: Various optimizer for single stage production. Naval Research Logistic Quarterly, 3:59-66, (1956).

[33] A. Tarim, S. Manandhar, T. Walsh: Stochastic constraint programming: A scenario-based approach. Constraints, 11:53-80, (2006).

[34] A. H. M. Ter Hofstede, M. Weske: Business process management: A survey. Proc. of the 1st International Conference on Business Process Management, LNCS 2678:1-12, (2003).

[35] I. Tsamardinos, M. E. Pollack, J. F. Horty: Merging Plans with Quantitative Temporal Constraints, Temporally Extended Actions, and Conditional Branches. Proc of the 5th International Conference on AI Planning Systems, 264-272, (2000).

[36] I. Tsamardinos, T. Vidal, M. E. Pollack: CTP: A New Constraint-Based Formalism for Conditional Temporal Planning. Constraints, 8(4):365-388, (2003).

[37] T. Vidal and H. Fargier: Handling Contingencies in Temporal Constraint Network: from Consistency to Controllability. Journal of Experimental and Theoretical Artificial Intelligence, 11:23-45, (1999).

[38] P. Vilim, R. Bartak, O. Cepek: Extension of o(n.log n) filtering algorithms for the unary resource constraint to optional activities. Constraints, 10:403–425, (2005).

[39] Y. Xie and W. Wolf: Allocation and scheduling of conditional task graph in hardware /software co-synthesis. Proc. of Design, Automation and Test in Europe, DATE2001, 620-625, (2001).

[40] T. Walsh: Stochastic constraint programming. Proc. of the European Conference on Artificial Intelligence, ECAI, (2002).

[41] M. Weske: Business Process Management: Concepts, Languages, Architectures, Springer Berlin, (2007).

[42] D. Wu, B. Al-Hashimi, P. Eles: Scheduling and mapping of conditional task graph for the synthesis of low power embedded systems. Computers and Digital Techniques, IEEE Proceedings. 150(5):262-273, (2003).

Appendix A: BFG construction procedure if CFU holds

The BFG construction procedure has exponential time complexity in case the CTG satisfies CFU since the number of possible conjunctions in the activation events is exponential; in practice we can devise a polynomial time algorithm if Control Flow Uniqueness holds. In this case we know the BFG contains an F-node for each condition outcome in the original CTG, plus an F root node;

therefore we can design an algorithm to build a BFG with an F-node for each condition outcome, and check at each step whether CFU actually holds; if a violation is encountered we return an error.

In the following, we suppose that each CTG node t_i is labeled with the set of condition outcomes in all paths from the root node to t_i (or "upstream conditions"); this can be easily done in polynomial time by means of a forward visit of the graph.

A polynomial time complexity BFG building procedure for graphs satisfying CFU is shown in Algorithm 4. The algorithm performs a forward visit of the Conditional Task Graph starting from the root node; as the visit proceeds the BFG is built and the CTG nodes are mapped to F-nodes. The acyclicity of the CTG ensures that whenever a CTG node t_i is visited, all F-nodes needed to map it have already been built.

Figure 15 shows an example of the procedure, where the input is a subgraph of the CTG from Figure 2A. Initially (Figure 15B) the L set only contains the root task t_5 and $V = \emptyset$. The CTG node t_5 is visited (line 5) and mapped to the pre-built F-node F_0. The `mapping_set` function at line 6 returns the set of F-nodes in the current BFG on which t_5 has to be mapped; details on how to compute this set will be given later. In the next step (Figure 15C) t_6 is processed and mapped on F_0; however, being t_6 a branch, a new B-node is built (line 9) and an F-node for each condition outcome (F_b, $F_{\neg b}$). In Figure 15D t_7 is visited and, similarly to t_6, a new B-node and two new F-nodes are built. Next, t_8 is visited and mapped on F_b (Figure 15E); similarly t_9 is mapped on $F_{\neg b}$, t_{10} on F_c and t_{11} on $F_{\neg c}$ (those steps are not shown in the figure). Finally, in Figure 15F and 15G, nodes t_{20} and t_{21} are visited; since the first is triggered by both the outcomes b and $\neg b$ it is mapped directly on F_0; t_{21} is instead an and node, whose main predecessor is t_{10}, therefore the `mapping_set` function maps it on the same F-node as t_{10}.

The set $F(t_i)$ for each CTG node is computed by means of a two-phase procedure. In first place an extended set of F-nodes is derived by a forward visit of the BFG built so far. The visit starts from the root F-node. At each step: (A) if an F-node is visited, then all its children are also visited; (B) if a B-node is visited, then each child nde is visited only if the corresponding condition outcome appears in the label of t_i (which reports the "upstream outcomes"). The CTG node t_i is initially mapped to all leaves reached by the visit; for example, with reference to Figure 15G, CTG node t_{21} is first mapped to $F_b, F_{\neg b}, F_c$.

In the second phase this extended set of F-nodes is simplified by recursively applying two simplification rules; in order to make the description clearer we temporarily allow CTG branch nodes to be mapped on B-nodes: B-node mappings, however, will be discarded at the end of the simplification process.

Algorithm 4 Building a BFG

```
1: input:  a Conditional Task Graph with all nodes labeled with the set of their
   upstream conditions
2: build the root F-node $F_0$
3: let $L$ be the set of nodes to visit and $V$ the one of visited nodes. Initially $L$
   contains the CTG root and $V = \emptyset$
4: while $L \neq \emptyset$ do
5:    pick the first node $t_i \in L$, remove $t_i$ from $L$
6:    let $F(t_i) = \texttt{mapping\_set}(\texttt{t}_\texttt{i})$ be the set of F-nodes $t_i$ has to be mapped on
7:    map $t_i$ on all F-nodes in $F(t_i)$
8:    if $t_i$ is a branch then
9:       build a B-node $B_i$
10:      build an F-node $F_{Out}$ for each condition outcome $Out$ of the branch
11:      connect each F-node in $F(t_i)$ to $B_i$
12:   end if
13:   add $t_i$ to $V$
14:   for all child node $t_j$ of $t_i$ do
15:      if all parent nodes of $t_j$ are in $V$, add $t_j$ to $L$
16:   end for
17: end while
```

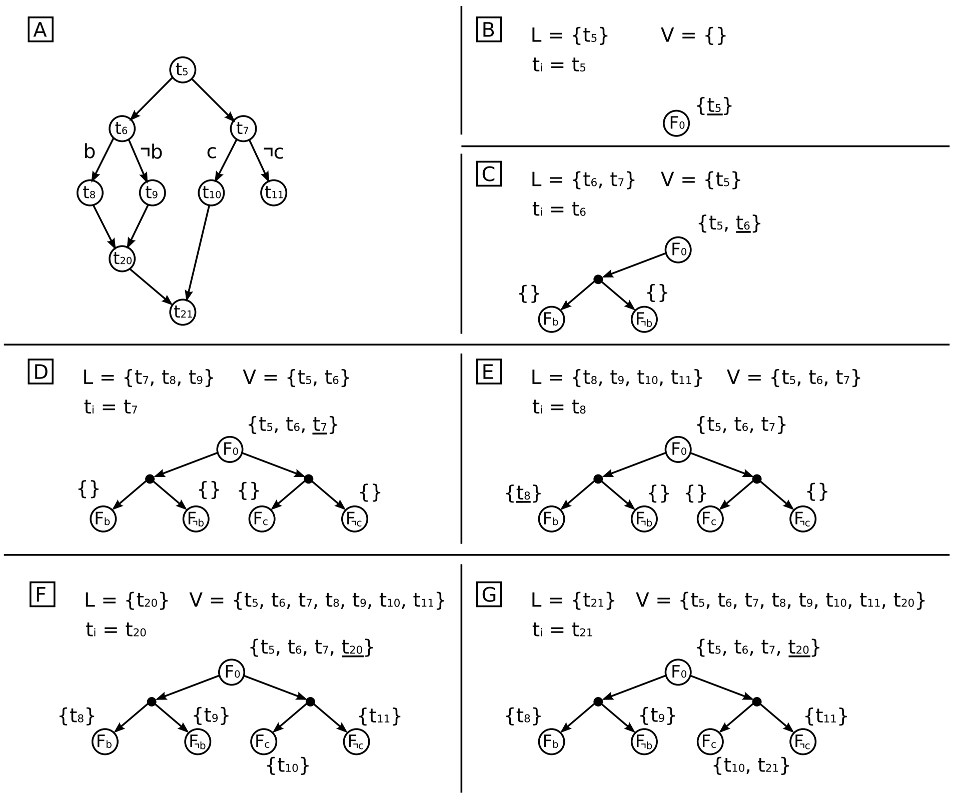


Fig. 15. BFG building procedure

rule 1: if a B-node B_i is the *only* parent of F-nodes $F_0, F_1, \ldots$ and a task t_j is mapped on all of them, then add B_i to $F(t_j)$. For example, with reference to Figure 15G, where initially $F(t_{21}) = \{F_b, F_{\neg b}, F_c\}$, after an application of rule 1 we have $F(t_{21}) = \{B_{left}, F_b, F_{\neg b}, F_c\}$

rule 2: if a task t_j is mapped on a B-node B_i with parent $F_0, F_1, \ldots$, then B_i and all its descendants can be removed from $F(t_j)$. If at the end of this process no descendant of F_0, F_1, etc is in $F(t_j)$, then map t_j on $F_0, F_1, etc.$ For example, after the application of rule 2, $F(t_{21})$ becomes $\{F_c\}$.

Once all simplifications are done, all remaining F-nodes in $F(t_i)$ must be mutually exclusive, as said in Section 5.1 and shown in Figure 8: if this fails to occur it means the BFG has not enough F-nodes for the mapping, which in turn means the original graph does not meet CFU. In this case an error is reported.

Appendix B: Improving the efficiency of the expected makespan constraint

In order to improve the computational efficiency of all filtering algorithms used in the expected makespan constraint (see Section 6.2.1), we can use F-nodes instead of tasks in the computation of $emkspan(S_{min})$ and $emkspan(S_{min})$. Remember that there is a mapping between tasks (CTG nodes) and F nodes. Each F-node can therefore be assigned a minimum and a maximum end value computed as follows:

$$\text{maxend}(F_j) = \max\{\max(end(t_i)) \mid t_i \in t(F_j)\}$$

$$\text{minend}(F_j) = \max\{\min(end(t_i)) \mid t_i \in t(F_j)\}$$

The rationale behind the formulas is that tasks mapped to an F-node F_i all execute in events in $\sigma(F_i)$; therefore the end time of the set of tasks will always be dominated by the one ending as last.

The two schedules S_{min}, S_{max} can store F-nodes (sorted by minend and maxend) instead of activities and their size can be reduced to at most $n_o + 1$ (where n_o is the number of condition outcomes, often $n_o \ll n_t$): this is in fact the number of F-nodes in a BFG if CFU holds (see Section 4.2).

Each time the end variable of a task t_i mapped to F_j changes, values maxend(F_j) and minend(F_j) are updated and possibly some nodes are swapped in S_{min}, S_{max} (similarly to what Figure 12B shows for tasks). These updates can be done with complexity $O(\max(n_t, n_o))$, where n_t is the number of tasks. The makespan bound calculation of constraints (6) can be done by substituting tasks with F-nodes in expression (5), as shown in expression 8:

$$E(makespan) = \sum_i p(F_i \wedge \neg F_{i+1} \wedge \ldots \wedge \neg F_{n_o-1})\; end(F_i) \qquad (8)$$

where $end(F_i) \in [minend(F_i), maxend(F_i)]$ and probabilities $p(F_i \wedge \neg F_{i+1} \wedge \ldots \wedge \neg F_{n_o-1})$ can be computed by querying the BFG with $q = F_i \wedge \neg F_{i+1} \wedge \ldots \wedge \neg F_{n_o-1}$. BFG queries involving F-nodes can be processed similarly to usual queries; basically, whilst each task is mapped to one or more F-nodes, an F node is always mapped to a *single* F-node (i.e. itself); thus, (a) the inclusion and exclusion labels can be computed as usual and (b) every update of the weight or the time window of an F-node is performed in strictly logarithmic time. The same algorithms devised for tasks can be used to prune the makespan and the end variables, but the overall complexity goes down to $O(\max(n_t, n_o^2 \log(n_o)))$.