Nested Temporal Networks with Alternatives

Roman Barták*, Ondřej Čepek*†

*Charles University, Faculty of Mathematics and Physics
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
{roman.bartak, ondrej.cepek}@mff.cuni.cz

†Institute of Finance and Administration
Estonská 500, 101 00 Praha 10, Czech Republic

Abstract

Temporal networks play a crucial role in modeling temporal relations in planning and scheduling applications. Recently, several extensions of temporal networks were proposed to integrate non-temporal information such as resource consumption or logical dependencies. Temporal Networks with Alternatives were proposed to model alternative and parallel processes, however the problem of deciding which nodes can be consistently included in such networks is NP-complete. In this paper we propose a tractable subclass of Temporal Networks with Alternatives that can still cover a wide range of real-life processes, while the problem of deciding node validity is solvable in polynomial time. We also present an algorithm that can effectively recognize whether a given network belongs to the proposed sub-class.

Introduction

Current temporal networks handle well temporal information including disjunction of temporal constraints (Stergiou and Koubarakis 1998) or uncertainty (Blythe, 1999). Several other extensions of temporal networks appeared recently such as resource temporal networks (Laborie 2003) or disjunctive temporal networks with finite domain constraints (Moffitt, Peintner, Pollack 2005). These extensions integrate temporal reasoning with reasoning on non-temporal information, such as fluent resources. All these approaches assume that all nodes are present in the network, though the position of nodes in time may be influenced by other than temporal constraints. Conditional Temporal Planning (Tsamardinos, Vidal, Pollack 2003) introduced an option to decide which node will be present in the solution depending on a certain external condition. Hence CTP can model conditional plans where the nodes actually present in the solution are selected based on external forces. In other problems, such as log-based reconciliation (Hamadi 2004), we need to model inter-dependencies between nodes which concern their presence/absence in the final solution. For example, the logical dependency $A \Rightarrow B$ used in log-based reconciliation problems says that if node A is present in the solution then node B must be present as well. The task is to select a subset of nodes that satisfy both logical and temporal constraints and respect some other constraints, (e.g. some nodes may be pre-selected to be present), or optimize certain objectives (e.g. maximize the number of selected nodes). The possibility to select nodes according to logical, temporal, and resource constrains was introduced to manufacturing scheduling by ILOG in their MaScLib (Nuijten at al. 2003). The same idea was independently formalized in Extended Resource Constrained Project Scheduling Problem (Kuster, Jannach, Friedrich 2007). In the common model each node has a Boolean validity variable indicating whether the node is selected to be in the solution. These variables are a discrete version of PEX variables used by Beck and Fox (1999) for modeling presence of alternative activities in the schedule. In many recent approaches, these variables are interconnected by logical constraints such as the dependency constraint described above. Recall that nodes usually correspond to activities (their start and/or end time) and the task is to allocate activities to time (and to resources), and also to decide which activities will actually be present in the solution. Hence, these frameworks are appropriate for modeling and solving over-subscribed scheduling problems or problems with alternative activities. Still, all these models handle logical and temporal constraints separately so they cannot take advantage of integrated reasoning similar to constraint filtering techniques proposed in (Barták and Čepek 2006). Temporal Networks with Alternatives (Barták and Čepek 2007) introduced a different type of alternatives with so called parallel and alternative branching. Temporal and logical constraints are closely integrated here – logical constraints are described as a part of branching in nodes. Unfortunately, the paper also showed that the problem of deciding which nodes can be consistently selected, if some nodes are pre-selected, is NP-hard. This result goes against a common-sense intuition which says that the selection of activities among several alternatives should be an easy task. This discrepancy may be caused by the fact that the presented model of Temporal Networks with Alternatives (TNA) is too general, while most real-life processes can be described using a very specific TNA.

In this paper we propose a restricted form of TNA which we call *Nested Temporal Networks with Alternatives*. This restriction is motivated by real-life manufacturing

Compilation copyright © 2007, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

scheduling problems where the network of alternatives has a specific topology. The main advantage of Nested TNA is the tractability of the assignment problem (decision about which nodes are valid). After a motivation example and recapitulation of TNA, we will formally define Nested TNA and present an algorithm that can recognize whether a given TNA is nested. The same algorithm (after a small extension) can also be used to decide which nodes can be present in the network, that is, to solve the TNA assignment problem (Barták and Čepek 2007). We conclude the paper by proposing new filtering rules for temporal relations that improve domain pruning proposed in (Barták and Čepek 2007).

Motivation and Background

Let us consider a manufacturing scheduling problem of piston production. Each piston consists of a rod and a tube that need to be assembled together to form the piston. Each rod consists of the main body and a special kit that is welded to the rod (the kit needs to be assembled before welding). The rod body is sawn from a large metal stick. The tube can also be sawn from a larger tube. Rod body, the kit, and tube must be collected together from the warehouse to ensure that their diameters fit. If the tube is not available, it can be bought from an external supplier. In any case some welding is necessary to be done on the tube before it can be assembled with the rod. Finally, between sawing and welding, both rod and tube must be cleared of metal cuts produced by sawing. Assume that welding and sawing operations require ten time units, assembly operation requires five time units, clearing can be done in two time units, and the material is collected from warehouse in one time unit. If the tube is bought from an external supplier then it takes fifty time units to get it. Moreover, tube and rod must cool-down after welding which takes five time units.

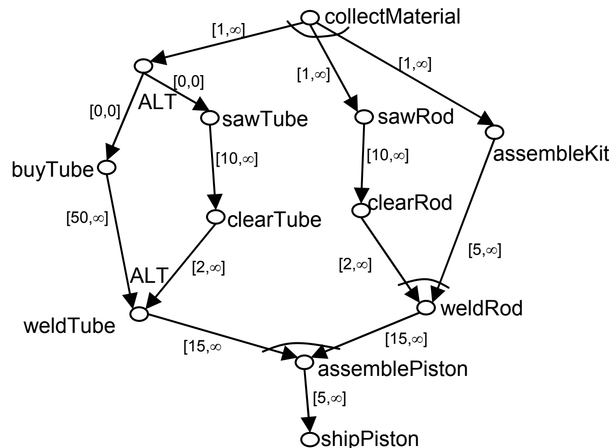


Figure 1. Example of a manufacturing process with alternatives.

The manufacturing processes from the above problem can be described using a Simple Temporal Network with Alternatives depicted in Figure 1. Nodes correspond to start times of operations and arcs are annotated by simple temporal constraints in the form [a, b], where a describes the minimal distance (in time) between the nodes and b describes the maximal distance. Informally, this network describes the traditional simple temporal constraints (Dechter, Meiri, Pearl 1991) together with the specification of branching of processes. There is a *parallel branching* marked by a semi-circle indicating that the process splits and runs in parallel and an *alternative branching* marked by ALT indicating that the process will consists of exactly one alternative path (we can choose between buying a tube and producing it in situ). We can see that this TNA has a very specific topology that we will try to address in the rest of the paper.

Let us now formally define Simple Temporal Networks with Alternatives from (Barták and Čepek 2007). Let G be a directed acyclic graph. A sub-graph of G is called a *fan-out sub-graph* if it consists of nodes x, $y_1,\ldots, y_k$ (for some k) such that each (x, y_i), $1 \le i \le k$, is an arc in G. If $y_1,\ldots, y_k$ are all and the only successors of x in G (there is no z such that (x, z) is an arc in G and $\forall i = 1,\ldots,k$: $z \ne y_i$) then we call the fan-out sub-graph complete. Similarly, a sub-graph of G is called a *fan-in sub-graph* if it consists of nodes x, $y_1,\ldots, y_k$ (for some k) such that each (y_i, x), $1 \le i \le k$, is an arc in G. A complete fan-in sub-graph is defined similarly as above. In both cases x is called a *principal node* and all $y_1,\ldots, y_k$ are called *branching nodes*.

Definition 1: A directed acyclic graph G together with its pair wise edge-disjoint decomposition into complete fan-out and fan-in sub-graphs, where each sub-graph in the decomposition is marked either as a *parallel* sub-graph or an *alternative* sub-graph, is called a *P/A graph*.

Definition 2: *Simple Temporal Network with Alternatives* is a P/A graph where each arc (X,Y) is annotated by a pair of numbers [a,b] where a describes the minimal distance between nodes X and Y and b describes the maximal distance, formally, $a \le$ Y-X $\le b$.

Figure 1 shows an example of Simple Temporal Network with Alternatives. If we remove the temporal constraints from this network then we get a P/A graph. Note that the arcs (*sawTube*, *clearTube*), (*sawRode*, *clearRod*), and (*assemblePiston*, *shipPiston*) form simple fan-in (or fan-out, it does not matter in this case) sub-graphs. As we will see later, it does not matter whether the sub-graphs consisting of a single arc are marked as parallel or alternative – the logical constraint imposed by the sub-graph will be always the same. Hence, we can omit the explicit marking of such single-arc sub-graphs to make the figure less overcrowded.

In this paper, we focus mainly on handling special logical relations imposed by the fan-in and fan-out sub-graphs – we call then *branching constraints*. Temporarily, we omit the temporal constraints, so we will work with P/A graphs only, but we will go back to temporal constraints later in the paper. In particular, we are interested in finding whether it is possible to select a subset of nodes in such a way that they form a feasible graph

according to the branching constraints. Formally, the selection of nodes can be described by an *assignment* of 0/1 values to nodes of a given P/A graph, where value 1 means that the node is selected and value 0 means that the node is not selected. The assignment is called *feasible* if

- in every parallel sub-graph all nodes are assigned the same value (both the principal node and all branching nodes are either all 0 or all 1),
- in every alternative sub-graph either all nodes (both the principal node and all branching nodes) are 0 or the principal node and exactly one branching node are 1 while all other branching nodes are 0.

Notice that the feasible assignment naturally describes one of the alternative processes in the P/A graph. For example, *weldRod* is present if and only if both *clearRod* and *assembleKit* are present (Figure 1). Similarly, *weldTube* is present if exactly one of nodes *buyTube* or *clearTube* is present (but not both). Though, the alternative branching is quite common in manufacturing scheduling, it cannot be described by binary logical constraints from MaScLib (Nuijten at al. 2003) or Extended Resource Constrained Project Scheduling Problem (Kuster, Jannach, Friedrich 2007). On the other hand, the branching constraints are specific logical relations that cannot capture all logical relations between the nodes.

It can be easily noticed that given an arbitrary P/A graph the assignment of value 0 to all nodes is always feasible. On the other hand, if some of the nodes are required to take value 1, then the existence of a feasible assignment is by no means obvious. Let us now formulate this decision problem formally.

Definition 3: Given a P/A graph G and a subset of nodes in G which are assigned to 1, *P/A graph assignment problem* is "Is there a feasible assignment of 0/1 values to all nodes of G which extends the prescribed partial assignment?"

Intuition motivated by real-life examples says that it should not be complicated to select the nodes to form a valid process according to the branching constraints described above. The following proposition from (Barták and Čepek 2007) says the opposite.

Proposition 1: The P/A graph assignment problem is NP-complete.

In the rest of the paper, we will propose a restricted form of the P/A graph, a so called nested P/A graph that can cover many real-life problems while keeping the P/A graph assignment problem tractable.

Nested P/A Graphs

When we analyzed how the P/A graphs modeling real-life processes look, we noticed several typical features. First, the process has usually one start point and one end point. Second, the graph is built by decomposing meta-processes into more specific processes until non-decomposable processes (operations) are obtained. There are basically two (three) types of decomposition. The meta-process can split into two or more processes that run in a sequence, that is, after one process is finished, the subsequent process can start. The meta-process can split into two or more sub-processes that run in parallel, that is, all sub-processes start at the same time and the meta-process is finished when all sub-processes are finished. Finally, the meta-process may consists of several alternative sub-processes, that is, exactly one of these sub-processes is selected to do the job of the meta-process. Notice, that the last two decompositions have the same topology of the network (Figure 2), they only differ in the meaning of the branches in the network. Note finally, that we are focusing on modeling instances of processes with particular operations that will be allocated to time. Hence we do not assume loops that are sometimes used to model abstract processes.

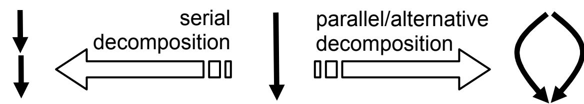


Figure 2. Possible decompositions of the process.

Based on above observations we propose a recursive definition of a nested graph.

Definition 4: A directed graph G = ($\{s,e\}$, $\{(s,e)\}$) is a *(base) nested graph*. Let G = (V, E) be a graph, $(x,y) \in$ E be its arc, and $z_1,\ldots, z_k$ ($k > 0$) be nodes such that neither z_i is in V. If G is a nested graph (and I = $\{1,\ldots,k\}$) then graph G' = (V $\cup$ $\{z_i \mid i\in$I$\}$, E $\cup$ $\{(x,z_i), (z_i,y) \mid i\in$I$\}$ $-$ $\{(x,y)\}$) is also a *nested graph*.

According to Definition 4, any nested graph can be obtained from the base graph with a single arc by repeated substitution of any arc (x,y) by a special sub-graph with k nodes (see Figure 3). Notice that a single decomposition rule covers both the serial process decomposition (k = 1) and the parallel/alternative process decomposition (k > 1). Though this definition is constructive rather than fully declarative, it is practically very useful. Namely, interactive process editors can be based on this definition so the users can construct only valid nested graphs by decomposing the base nested graph.

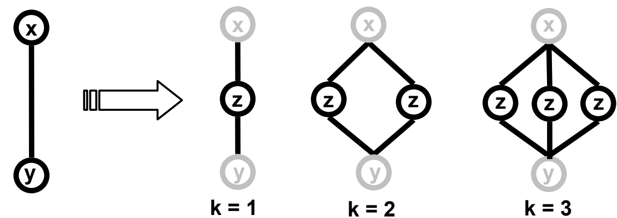


Figure 3. Arc decomposition in nested graphs.

The directed nested graph defines topology of the nested P/A graph but we also need to annotate all fan-in and fan-

out sub-graphs as either alternative or parallel sub-graphs. Moreover, we need to do the annotation carefully so the assignment problem can be solved easily for nested graphs and no node is inherently invalid. The idea is to annotate each node by input and output label which defines the type of branching (fan-in or fan-out sub-graph).

Definition 5: *Labeled nested graph* is a nested graph where each node has (possibly empty) input and output labels defined in the following way. Nodes s and e in the base nested graph and nodes z_i introduced during decomposition have empty initial labels. Let k be the parameter of decomposition when decomposing arc (x,y). If $k > 1$ then the output label of x and the input label of y are unified and set either to PAR or to ALT (if one of the labels is non-empty then this label is used for both nodes).

Figure 4 demonstrates how the labeled nested graph is constructed for the motivation example from Figure 1. In particular, notice how the labels of nodes are introduced (a semicircle for PAR label and A for ALT label). When a label is introduced for a node, it never changes in the generation process.

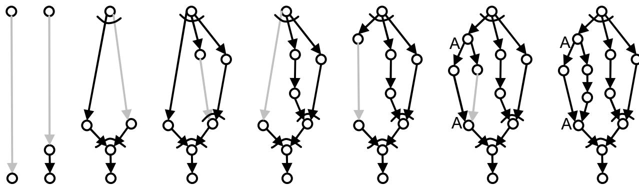


Figure 4. Building a labeled nested graph.

If an arc (x, y) is being decomposed into a sub-graph with k new nodes where $k > 1$, then we require that the output label of x is unified with the input label of y. This can be done only if either both labels are identical or at least one of the labels is empty. The following lemma shows that the second case always holds.

Lemma 1: For any arc (x, y) in the labeled nested graph, either the output label of x or the input label of y is empty.

Proof: The base nested graph contains a single arc (s, e) and labels for s and e are empty so the arc (the graph) satisfies the lemma. Assume now that graph G satisfies the lemma and we decompose some arc (x, y). During the decomposition, arc (x, y) is removed from the graph and substituted by arcs (x, z_i) and (z_i, y) for new nodes z_i, $1 \leq i \leq k$, which have empty labels. Hence, the new arcs satisfy the lemma. According to Definition 5 if $k > 1$ the output label of x and the input label of y are set (both or just one of them, if the other one was set already) so we need to check the other arcs going from x or going to y. If there was another arc (x, b) in G in addition to removed (x, y) then some arc (x, c) has already been decomposed to obtain two or more arcs going from x. Hence the output label of x has already been set in G and according to assumption the input label of b was empty which is preserved in the new graph. Symmetrically, if there was additional arc (b, y) in G then the output label of b is empty. So, all arcs in graph G that remain in the new graph still satisfy the lemma. □

Now, we can formally introduce a nested P/A graph.

Definition 6: *A nested P/A graph* is obtained from a labeled nested graph by removing the labels and defining the fan-in and fan-out sub-graphs in the following way. If the input label of node x is non-empty then all arcs (y, x) form a fan-in sub-graph which is parallel for label PAR or alternative for label ALT. Similarly, nodes with a non-empty output label define fan-out sub-graphs. Each arc (x, y) such that both output label of x and input label of y are empty forms a parallel fan-in sub-graph.

Note, that requesting a single arc to form a parallel fan-in sub-graph is a bit artificial. We use this requirement to formally ensure that each arc is a part of some sub-graph.

Proposition 2: A nested P/A graph is a P/A graph.

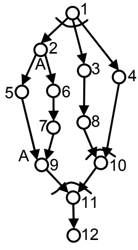
Proof: A nested P/A graph is a directed acyclic graph because the base nested graph is acyclic and the decomposition rule does not add a cycle. From Lemma 1, for each arc (x, y) either the output label of x or the input label of y is empty If both labels are empty then the arc forms a separate fan-in sub-graph. If the input label of x is non-empty then the arc belongs to a fan-out sub-graph with principal node x. Similarly, if the output label of y is non-empty then the arc belongs to a fan-in sub-graph with principal node y. Consequently, each arc belongs to exactly one sub-graph so the nested P/A graph is a P/A graph. □

Recognizing Nested P/A Graphs

Proposition 2 claims that a nested P/A graph is a special form of a P/A graph. It is easy to show that there exist P/A graphs which are not nested (see Conclusions). Hence, an interesting question is whether we can efficiently recognize whether a given P/A graph is nested. In this section we will present a polynomial algorithm that can recognize nested P/A graphs by reconstructing how they are built.

First, notice that in a nested P/A graph there are no two different fan-in (fan-out) sub-graphs sharing the same principal node (Definition 6). In other words, either all arcs going to (from) a given node x belong to a single fan-in (fan-out) sub-graph with the principal node x or there is no fan-in (fan-out) sub-graph with that principal node. This feature is easy to detect so in the rest of the paper, we assume that each node participates as a principal node in at most one fan-in and at most one fan-out sub-graph. This is reflected in the following representation of P/A graphs (Figure 5). The P/A graph is represented as a set of nodes where each node x is annotated by sets of predecessors *pred*(x) and successors *succ*(x) in the graph and by labels *inLab*(x) and *outLab*(x). *inLab*(x) = PAR if x is a principal node in a fan-in parallel sub-graph, *inLab*(x) = ALT if x is a principal node in a fan-in alternative sub-graph. If x is not a principal node in any fan-in sub-graph then *inLab*(x) is empty. A similar definition is done for *outLab*(x) with relation to fan-out sub-graphs. Notice the similarity of labels to labeled nested graphs (Definition 5). The reader

should realize that any nested P/A graph can be represented this way: all fan-in and fan-out sub-graphs correspond to non-empty labels and for any arc (x, y) either the label $outLab(x)$ or $inLab(y)$ is empty.



node	*pred*	*succ*	*inLab*	*outLab*
1		2,3,4	-	PAR
2	1	5,6	-	ALT
3	1	8	-	-
4	1	10	-	-
5	2	9	-	-
6	2	7	-	-
7	6	9	PAR	-
8	3	10	PAR	-
9	5,7	11	ALT	-
10	4,8	11	PAR	-
11	9,10	12	PAR	-
12	11		PAR	-

Figure 5. Representation of a (nested) P/A graph.

The following algorithm *DetectNested* recognizes labeled nested graphs by reconstructing how they are built.

```
algorithm DetectNested(input: graph G, output: {success, failure})
1. select all nodes x in G such that |pred(x)| = |succ(x)| = 1
2. sort the selected nodes lexicographically according to index
        (pred(x), succ(x)) to form a queue Q
3. while non-empty Q do
4.  |  select and delete a sub-sequence L of size k in Q such that
    |      all nodes in L have an identical index ({x}, {y}) and
    |      either |succ(x)| = k or |pred(y)| = k
5.  |  if no such L exists then stop with failure
6.  |  if k > 1 & outLab(x) ≠ inLab(y)  then stop with failure
7.  |  remove nodes z∈L from the graph
8.  |  remove nodes x, y from Q (if they are there)
9.  |  add arc (x,y) to the graph (an update succ(x) and pred(y))
10. |  if |pred(x)| =  |succ(x)| = 1 then insert x to Q
11. |  if |pred(y)| =  |succ(y)| = 1 then insert y to Q
12.end while
13.if the graph consists of two nodes connected by an arc then
14.    stop with success
15.else stop with failure
```

Proposition 3: Algorithm *DetectNested* always terminates and it stops with success if and only if the input P/A graph is nested.

Proof: Each line of the algorithm terminates. The body of the while loop either terminates with a failure or at least one node is removed from the graph. Because the queue Q consists of nodes that are part of the current graph, it must become empty sometime so the while loop terminates and hence the whole algorithm terminates.

We will show that the algorithm recognizes labeled nested graphs by induction on the number of decomposition steps necessary to generate a graph. The base nested graph is trivially recognized in line 13. Assume now that the algorithm can recognize all nested graphs built using m steps. We shall show that:

(i) if *DetectNested* fails to find a set L of nodes to be contracted then the input graph is not a labeled nested graph, and

(ii) if *DetectNested* finds a set L of nodes and contracts them and the input graph is a labeled nested graph build using $(m+1)$ steps, then the resulting graph is a labeled nested graph which can be built using m steps.

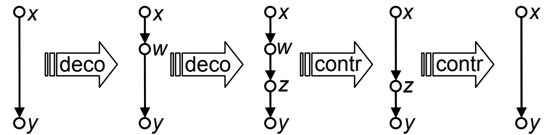
It is easy to see that these two claims are sufficient for the proof of the equivalence part of the proposition.

To prove (i) it is enough to realize that in any labeled nested graph constructed in accordance with Definition 4, the nodes added in the last decomposition step always fulfill the requirements on the set L in *DetectNested*. Thus if *DetectNested* fails to find a suitable set L then the input graph is not a labeled nested graph.

The proof of (ii) is more difficult because of the fact that there may be many suitable sets L = $\{ z_1, \ldots, z_k \}$ which *DetectNested* may find and contract. We have to show that any such choice produces a graph, which is labeled nested and can be built using m steps. Let us consider two cases:

a) $k > 1$. In this case a parallel or alternative sub-graph with nodes x, y, $z_1, \ldots, z_k$ and arcs (x, z_i), (z_i, y) is contracted into arc (x, y). Notice, that (using the assumption that the input graph is nested) this sub-graph must be a result of an arc decomposition of (x, y) during the recursive construction and moreover no arc inside this sub-graph is further decomposed. Therefore the graph which is obtained from the input graph by contraction of L is a labeled nested graph obtainable in m steps. The sequence of decomposition steps is the same as for the input graph except that the decomposition of (x, y) is skipped.

b) $k = 1$. In this case a chain of length ≥ 2 is shortened (by one vertex and one arc) by the contraction of L. In this case there is no guarantee that the contraction can be matched to a decomposition step which built the input graph (see example below).



However, the chain of length l can be produced from a single arc by $(l-1)$ decompositions and can be contracted back into a single arc by $(l-1)$ contractions in *DetectNested* (all with $|L| = 1$). Thus, similarly as in case a) the graph which is obtained from the input graph by contraction of L is a labeled nested graph obtainable in m steps. The sequence of decomposition steps is the same as for the input graph except that the sub-sequence (not necessarily a sub-interval) of $(l-1)$ decompositions which built the chain is replaced by $(l-2)$ decompositions which build the shorter chain. □

Proposition 4: The worst-case time complexity of algorithm *DetectNested* is $O(n^2)$, where n is a number of nodes in the graph.

Proof: The initial selection of nodes for the queue can be done in time $O(n)$. Time $O(n.\log n)$ is necessary to sort the queue. The sub-list for contraction can be selected in time $O(n)$ and insertion of nodes into the list can be done in $O(n)$. All other operations can be implemented in constant

time. The while loop is repeated at most n times because each time at least one node is removed from the graph. Together, the while loop takes time $O(n^2)$ so the whole algorithm takes time $O(n^2)$. □

Tractability of Nested P/A Graphs

The main motivation for introducing nested P/A graphs was to make the P/A graph assignment problem tractable for this special group of graphs. Recall that the assignment problem consists of deciding whether it is possible to complete a partial assignment of validity variables for nodes to obtain a complete feasible assignment. We can reformulate the P/A graph assignment problem as a constraint satisfaction problem in the following way. Each node x is represented using a Boolean validity variable v_x, that is a variable with domain {0,1}. If the arc between nodes x and y is a part of some parallel sub-graph then we define the following constraint:

$$v_x = v_y.$$

If x is a principal node and $y_1,\dots, y_k$ for some k are all branching nodes in some alternative sub-graph then the logical relation defining the alternative branching can be described using the following arithmetic constraint:

$$v_x = \Sigma_{j=1,\dots,k}\, v_{y_j}.$$

Notice that if $k = 1$ then the constraints for parallel and alternative branching are identical (hence, it is not necessary to distinguish between them). Notice also that the arithmetic constraint for alternative branching together with using the {0,1} domains defines exactly the logical relation between the nodes – v_x is assigned to 1 if and only if exactly one of v_{y_j} is assigned to 1. Using the arithmetic constraint simplifies a formal description of the relation and also simplifies the proof of the following proposition. The task whether a completion of the partial assignment of validity variables exists is clearly equivalent to the assignment problem for the original P/A graph. We will show now that the contraction operations of the *DetectNested* algorithm can be described as simple arithmetic operations over the above constraint model describing the nested P/A graph which leads to solving the P/A graph assignment problem.

Proposition 5: The assignment problem for a nested P/A graph is tractable (can be solved in polynomial time).

Proof: We will show how to find a feasible assignment for a nested P/A graph, if it exists, or to prove that no assignment exists. The main idea is to use the *DetectNested* algorithm from the previous section.

The contraction operation of the *DetectNested* algorithm is a reverse operation to the decomposition operation used when building the nested graph (Figure 3). Let us assume that a sub-graph with nodes x, y, z_1, …, z_k and arcs (x, z_i), (z_i, y) is being contracted into arc (x, y). The contraction operation is allowed if and only if z_1, …, z_k are either all successors of x or all predecessors of y (line 4 of the algorithm) and the type of fan-out graph with the principal node x is identical to the type of fan-in graph for y (line 6) (if $k = 1$ then the equality of types does not matter as we mentioned above hence it is not requested in the contraction algorithm). We also know that no node z_i appears elsewhere in the graph. Without lost of generality, let us assume that z_i are all predecessors of y and z_j', $j = 1,\dots, m$ ($m \geq 0$) are the remaining successors of x. We distinguish two cases: parallel and alternative branching. In the case of parallel branching, the P/A graph assignment problem before contraction is described using constraints C, $x = z_i$, $x = z_j$', $z_i = y$, where C is the set of constraints for the rest of the graph (note that C does not contain z_i). The problem after contraction is described by constraints C, $x = y$, $x = z_j$'. These two sets of constraints are visibly equivalent, meaning that a solution for constraints before contraction exists if and only if a solution for constraints after contraction exists. In the case of alternative branching, we have the constraints C, $x = \Sigma_{i=1,\dots,k}\, z_i + \Sigma_{j=1,\dots,m}\, z_j$', $\Sigma_{i=1,\dots,k}\, z_i = y$ before contraction and constraints C, $x = y + \Sigma_{j=1,\dots,m}\, z_j$' after contraction. Again, these sets of constraints are equivalent. Assume now that a partial assignment of variables is given. If neither z_i is part of the assignment then we can contract the graph, find a feasible assignment for the contracted graph (if it exists) and then extend this assignment to variables z_i. If any z_i is part of the partial assignment then we check whether the removed constraint(s), either $z_i = y$ or $\Sigma_{i=1,\dots,k}\, z_i = y$, can be satisfied. If the answer is yes then we compute the (unique) value for y (and values for not-yet instantiated z_i), add it to the partial assignment, and continue as before. Otherwise no feasible assignment exists and the algorithm can stop. To finish the proof, one needs to realize that for a nested P/A graph the algorithm stops with a single arc (s, e) and the assignment problem can be trivially solved for this graph. □

Temporal Constraints

So far, we ignored the temporal part of Simple Temporal Networks with Alternatives and we focused merely on logical relations imposed by the branching constraints. The reason was to show that logical reasoning is easy for nested P/A graphs (while it is hard for general P/A graphs even without temporal relations). Now we can return back the simple temporal constraints. Notice that the selected feasible set of nodes together with arcs between them forms a sub-graph of the original P/A graph. We require this sub-graph to be also *temporally feasible*, which means that all the simple temporal constraints between the valid nodes are satisfied in the sense of simple temporal networks (Dechter, Meiri, Pearl 1991). Naturally, the logical and temporal reasoning is interconnected – if a temporal constraint between nodes x and y cannot be satisfied then (at least) one of the nodes must be invalid (it is assigned to 0). Formally, we can extend the constraint model introduced in the previous section by annotating each node i by a temporal variable t_i indicating the position of the node in time. For simplicity reasons we assume that the domain of such variables is an interval $\langle 0, MaxTime\rangle$ of integers, where *MaxTime* is a constant given by the user.

Recall that the temporal relation between nodes i and j is described by a pair $[a_{i,j}, b_{i,j}]$. This relation can now be naturally represented using the following constraint:

$$v_i * v_j * (t_i + a_{i,j}) \leq t_j \;\wedge\; v_i * v_j * (t_j - b_{i,j}) \leq t_i.$$

If $b_{i,j} = \infty$ then the second part of conjunction is omitted and similarly if $a_{i,j} = -\infty$ then the first part of conjunction is omitted. Notice that if any v_i or v_j equals zero, that is, some involved node is invalid, then the constraint is trivially satisfied (we get $0 \leq t_j \;\wedge\; 0 \leq t_i$). If both v_i and v_j equal 1 then we get ($t_i + a_{i,j} \leq t_j \;\wedge\; t_j - b_{i,j} \leq t_i$), which is exactly the simple temporal relation between nodes i and j. Figure 6 shows how the domains from the previous example (Figure 1) will look after filtering out the infeasible values by making the above constraint model arc consistent. We assume that *shipPiston* (the bottom node) is a valid node and *MaxTime* = 70. Black nodes are valid; validity of white nodes is not decided yet. Notice weak domain pruning of time variables in the white nodes caused by a disjunctive character of the problem. Actually, the left most path (with *buyTube*) cannot be selected due to time constraints but this is not discovered by making the constraints arc consistent.

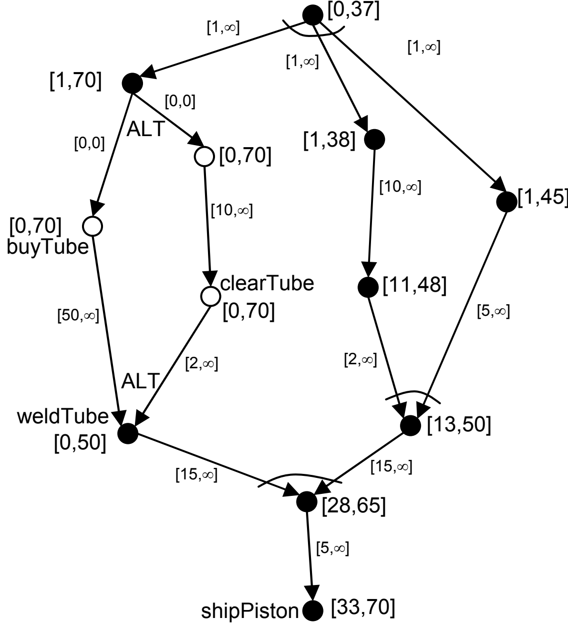


Figure 6. Domain filtering using the proposed constraint model.

The above described temporal constraints with validity variables do not take in account information about the type of branching. In (Barták and Čepek 2006) we showed that integrating logical and temporal reasoning can significantly improve domain pruning. In particular, we can take in account information about relations between the validity variables when propagating the temporal constraint. We propose to always propagate the temporal constraint, that is, to prune domains of temporal variables. If the temporal constraint is violated then, instead of generating a failure, we set some validity variable to 0. We shall describe the proposed filtering rules in more detail now. Note that the filtering rule propagates changes of domains between the constrained variables, namely, the values that violate the constraint are removed from the domains. Let $d(x)$ by a domain of variable x. In our model, the domain of a temporal variable is an interval so we can use interval arithmetic to propagate the temporal constraints. Namely, $\langle l, u\rangle + \langle a, b\rangle = \langle l+a, u+b\rangle$ and $\langle l, u\rangle - \langle a, b\rangle = \langle l-b, u-a\rangle$. Assume that arc (i, j) is a part of parallel branching, so in the solution either both nodes are valid and the temporal relation must hold, or both nodes are invalid and the temporal relation does not play any role (the domains of temporal variables are irrelevant provided that they are non-empty). Hence, we can always propagate the temporal relation provided that we properly handle its violation. Let $UP = d(t_j) \cap (d(t_i) + \langle a_{i,j}, b_{i,j}\rangle)$. The following filtering rule is applied whenever $d(t_i)$ changes:

$$d(t_j) \leftarrow UP \qquad \text{if } UP \neq \emptyset$$
$$d(v_j) \leftarrow d(v_j) \cap \{0\} \qquad \text{if } UP = \emptyset.$$

Note that $UP = \emptyset$ means violation of the temporal relation which is accepted only if the nodes are invalid. If the nodes are valid then a failure is generated because the above rule makes the domain of the validity variable empty. Symmetrically, let $DOWN = d(t_i) \cap (d(t_j) - \langle a_{i,j}, b_{i,j}\rangle)$. The following filtering rule is applied whenever $d(t_j)$ changes:

$$d(t_i) \leftarrow DOWN \qquad \text{if } DOWN \neq \emptyset$$
$$d(v_i) \leftarrow d(v_i) \cap \{0\} \qquad \text{if } DOWN = \emptyset.$$

The following example demonstrates the effect of above filtering rules. Assume that the initial domain of temporal variables is $\langle 0, 70\rangle$, the validity of nodes is not yet decided, and there are arcs (i, j) and (j, k) with temporal constraints [10, 30] and [20, 20] respectively. The original constraints do not prune any domain, while our extended filtering rules set the domains of temporal variables t_i, t_j, and t_k to $\langle 0, 40\rangle$, $\langle 10, 50\rangle$, and $\langle 30, 70\rangle$ respectively. If the initial domain is $\langle 0, 20\rangle$ then the original constraints again prune nothing, while our extended filtering rules deduce that the participating nodes are invalid (we assume that logical constraints in the form $v_x = v_y$ are also present).

The propagation of temporal constraints in the alternative branching is more complicated because we need to propagate them together. Let x be the principal node of a fan-in alternative sub-graph and $y_1, \ldots, y_k$ be all branching nodes. We first show how the information from the branching nodes is propagated to the principal node. Let $UP = d(t_x) \cap \bigcup_{j=1,\ldots,k} \{ (d(t_{y_j}) + \langle a_{y_j,x}, b_{y_j,x}\rangle) \mid d(v_{y_j}) \neq \{0\} \}$. The following filtering rule is applied whenever any $d(t_{y_j})$ or $d(v_{y_j})$ changes:

$$d(t_x) \leftarrow UP \qquad \text{if } UP \neq \emptyset$$
$$d(v_x) \leftarrow d(v_x) \cap \{0\} \qquad \text{if } UP = \emptyset.$$

In the alternative branching we do not know which arc is used in the solution so we need to assume them all. Therefore, the above rule uses a union of pruned domains proposed by individual arcs (from non-invalid nodes). Symmetrically, let $DOWN_j = d(t_{y_j}) \cap (d(t_x) - \langle a_{y_j,x}, b_{y_j,x}\rangle)$. The following filtering rule is applied to all t_{y_j} whenever

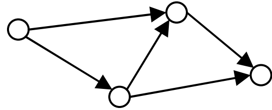
$d(t_x)$ changes and the change is propagated to branching nodes.

$$d(t_{y_j}) \leftarrow DOWN_j \qquad \text{if } DOWN_j \neq \emptyset$$
$$d(v_{y_j}) \leftarrow d(v_{y_j}) \cap \{0\} \qquad \text{if } DOWN_j = \emptyset.$$

Similar filtering rules can be designed for fan-out alternative sub-graphs. Again, the main advantage of these rules is stronger pruning in comparison with the original constraints as we shall show using the example from Figure 6. In particular, if we propagate from *weldTube* to *buyTube* and *clearTube*, we obtain $\langle 0, 0\rangle$ and $\langle 0, 48\rangle$ as new domains of corresponding temporal variables. Now, if we propagate through the alternative branching going to *buyTube*, we deduce that this node is invalid because the corresponding temporal constraint is violated. Consequently, all remaining nodes are valid.

Conclusions

The paper proposes a recursive definition of temporal networks with alternative processes, so called nested temporal networks with alternatives. The definition is motivated by a structure of typical manufacturing processes that consists either of a sequence of serial sub-processes or of a set of concurrent sub-processes which are either parallel or alternative. This model is general enough to cover many real-life processes, though we are aware that there exist reasonable processes that are not covered such as the following process.



We used the framework of Temporal Networks with Alternatives (TNA) to formally describe our idea and we also proved that the problem of selecting nodes to form a valid process, that is, selecting nodes satisfying the inherent logical dependencies, is tractable for the nested subclass (while this problem is NP-complete for general TNA). We also proposed an efficient algorithm for recognizing nested temporal networks. Finally, we presented preliminary filtering rules for temporal constraints with validity variables. Though we have no formal proof yet, we believe that the proposed extension does not harm tractability of simple temporal constraints provided that the network is nested. The background of this paper is in the area of temporal networks that are frequently used to model instances of processes. There also exists a large research area of modeling abstract processes using Petri Nets (van der Aalst 1998) which may serve as a source of other generalizations of temporal networks.

Acknowledgements

The research is supported by the Czech Science Foundation under the contract no. 201/07/0205 and by the EMPOSME project under EU FP6 scheme.

References

W.M.P. van der Aalst. 1998. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21-66.

Barták, R.; Čepek, O. 2006. Incremental Filtering Algorithms for Precedence and Dependency Constraints. In *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2006)*, 416-423. IEEE Press.

Barták, R. and Čepek, O. 2007. Temporal Networks with Alternatives: Complexity and Model. In *Proceedings of the Twentieth International Florida AI Research Society Conference (FLAIRS 2007)*. AAAI Press.

Beck, J.Ch. and Fox, M.S. 1999. Scheduling Alternative Activities. *Proceedings of AAAI-99*, 680-687, AAAI Press.

Blythe, J. 1999. An Overview of Planning Under Uncertainty. *AI Magazine,* 20(2): 37–54.

Dechter, R.; Meiri, I. and Pearl, J. 1991. Temporal Constraint Networks. *Artificial Intelligence*, 49:61-95.

Focacci, F.; Laborie, P.; and Nuijten, W. 2000. Solving Scheduling Problems with Setup Times and Alternative Resources. In *Proceedings of AIPS 2000*.

Hamadi, Y. 2004. Cycle-cut decomposition and log-based reconciliation. In *ICAPS Workshop on Connecting Planning Theory with Practice*, 30-35.

Kuster, J.; Jannach, D.; Friedrich, G. 2007. Handling Alternative Activities in Resource-Constrained Project Scheduling Problems. In *Proceedings of Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, 1960-1965.

Laborie, P. 2003. Resource temporal networks: Definition and complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 948-953.

Moffitt, M. D.; Peintner, B.; and Pollack, M. E. 2005. Augmenting Disjunctive Temporal Problems with Finite-Domain Constraints. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-2005)*, 1187-1192. AAAI Press.

Nuijten, W.; Bousonville, T.; Focacci, F.; Godard, D.; Le Pape, C. 2003. MaScLib: Problem description and test bed design, http://www2.ilog.com/masclib

Stergiou, K., and Koubarakis, M. 1998. Backtracking algorithms for disjunctions of temporal constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, 248-253. AAAI Press.

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