Modelling Alternatives in Temporal Networks

Roman Barták, Ondřej Čepek, and Pavel Surynek

***Abstract*—Temporal Networks play an important role in solving planning problems and they are also used, though not as frequently, when solving scheduling problems. In this paper we propose an extension of temporal networks by parallel and alternative branching. This extension supports modelling of alternative paths in the network; in particular, it is motivated by modelling alternative process routes in manufacturing scheduling. We show that deciding which nodes can be consistently included in this extended temporal network is an NP-complete problem. To simplify solving this problem, we propose a pre-processing step whose goal is to identify classes of equivalent nodes. The ideas are presented using precedence networks, but we also show how they can be extended to simple temporal networks.**

I. Introduction

SCHEDULING problems typically deal with allocating known activities to available resources and time. Real-life problems are usually more complex than existing theoretical models and, for example, they also require selection among alternative process routes or alternative resources in complex manufacturing enterprises. Due to efficiency issues, selection of alternative processes and resource allocation are frequently done separately from scheduling. However, this approach has several drawbacks. First, if the selected route or resource allocation cannot be scheduled, it is necessary to backtrack from the scheduling module to resource allocation and process selection modules. Second, even if the resource allocation and selected routes are feasible, separating the allocation and process selection algorithms from the scheduling algorithm may ruin the quality of the solution. Hence, a better result will be obtained when process selection and resource allocation is done within scheduling. While resource allocation is now an accepted part of scheduling problems and there exist approaches for doing resource allocation within scheduling, for example [5], process selection is still treated separately.

In this paper we propose an extension of temporal networks that can model alternative process routes. We describe the main ideas using networks with only precedence relations, but at the end of the paper we also show how these ideas can be extended to simple temporal networks. To model selection of alternatives, we assign a validity variable to each node in the network. This validity variable indicates whether the node is selected or not to be in the final solution plan. Decision about validity/invalidity of the node is done by the solver. We also augment the precedence network by a description of splitting and joining operations that implicitly define logical dependencies between nodes in the network. The dependency relations specify which nodes must/cannot be valid in relation to the validity status of other nodes. The main motivation for these operations goes from modelling manufacturing processes. The nodes correspond to activities (or more precisely start times of activities) and the arcs describe flow of products between the activities (precedence constraints). In some nodes the manufacturing process can split into two or more parallel sub-processes that can join back to a single process. For example, a piece of wood is cut in parts that are processed in parallel and then assembled together to a final product. This is called *parallel branching* (Figure 1 top with semicircle). Another form of branching is *alternative branching* when the process also splits in sub-processes, but these sub-processes are treated as alternatives so exactly one of them is used. The alternative sub-processes can also join back to a single process (Figure 1 bottom).

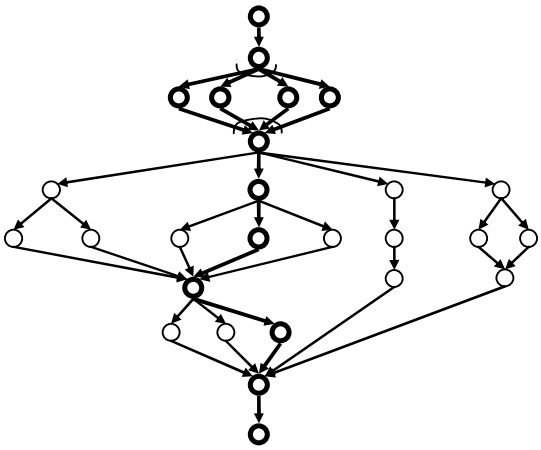


Fig. 1. Example of parallel (top) and alternative (bottom) sub-processes with one selected process.

In the paper we formally define the above-described precedence graph with parallel and alternative branching that we call a *P/A graph*. We also show that the problem whether there exists an assignment of validity variables consistent with the specified branching is NP-complete. To support

Manuscript received October 31, 2006. This work is supported by the Czech Science Foundation under the contract no. 201/04/1102 and by the EMPOSME project under EU FP6 scheme.

R. Barták is with Charles University, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic (phone: +420221914242 fax: +420221914323 e-mail: roman.bartak@mff.cuni.cz).

O. Čepek is with Charles University, Faculty of Mathematics and Physics and with Institute of Finance and Administration, Estonská 500, 101 00 Praha 10, Czech Republic (ondrej.cepek@mff.cuni.cz).

P. Surynek is with Charles University, Faculty of Mathematics and Physics (pavel.surynek@mff.cuni.cz).

problem solving, we propose a pre-processing step where sets of equivalent nodes are identified in the P/A graph. We call the nodes equivalent, if their validity status is identical in all solutions, that is, such nodes are either all valid or all invalid in any consistent assignment of validity variables. This will help us to bridge alternative routes. We conclude the paper by showing that the presented ideas can be extended to simple temporal networks. We compare our proposal with existing works on temporal networks. Namely, we demonstrate that P/A simple temporal networks can model temporal constraint satisfaction problems.

II. P/A Graphs

Let G be a directed acyclic graph. A subgraph of G is called a *fan-out subgraph* 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. Similarly, a subgraph of G is called a *fan-in subgraph* 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. 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 together with a set of its pairwise edge-disjoint fan-out and fan-in subgraphs, where each subgraph in the set is marked either as a *parallel* subgraph or an *alternative* subgraph, is called a *P/A graph*.

An *assignment* of 0/1 (true/false) values to nodes of a given P/A graph is called *feasible* if

- in every parallel subgraph 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 subgraph 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.

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 (as we shall see later, this requirement is a very natural one if the P/A graph is used to model a real-life problem), then the existence of a feasible assignment is by no means obvious. Let us now formulate this decision problem formally.

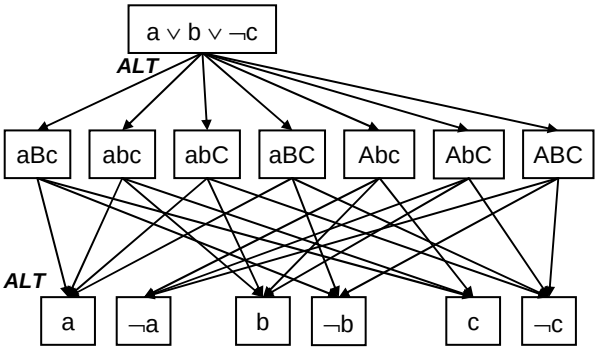
Definition 2: *P/A graph assignment problem* is given by a P/A graph G and a list of nodes of G which are assigned value 1. The question is whether there exist a feasible assignment of 0/1 values to all nodes of G which extends the prescribed partial assignment.

Remark: The P/A graph assignment problem remains the same if we allow forcing value 1 for just a single vertex. To see this, observe that the general case can be reduced to this special one by adding an extra vertex, forcing it to 1 and connecting it by a fan-in (or fan-out) parallel subgraph to all nodes that were forced to 1 originally. Moreover, if the original graph was acyclic, then so is the new one.

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

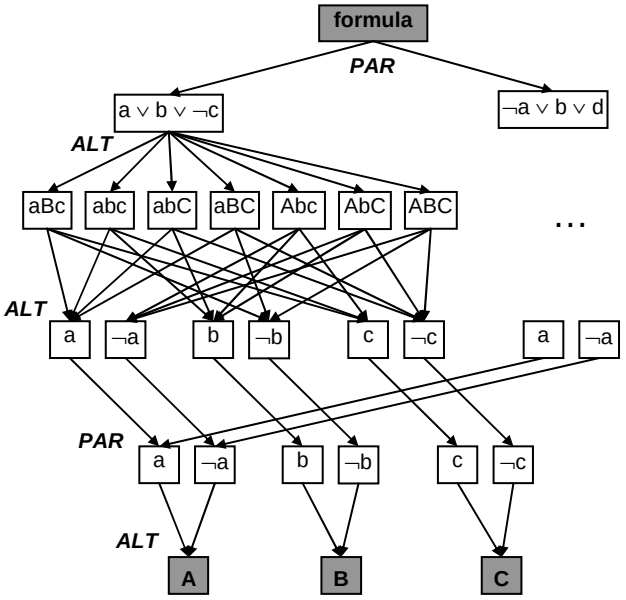
Proof: The problem is obviously in NP, because it suffices to guess the assignment and test its feasibility, which can be done in linear time in the number of parallel and alternative subgraphs (and hence in the number of edges). For the NP-hardness, we shall show that the 3SAT problem, which is known to be NP-complete [6], can be reduced (in a polynomial time) to the P/A graph assignment problem. Recall that the 3SAT problem is a problem of deciding whether there exists a model (a satisfying assignment of truth values to propositional variables) for a given formula in a conjunctive normal form, where each clause in the formula consists of exactly three literals. Moreover we may assume that no variable appears twice in a single clause, that is, each clause consists of literals of three distinct variables.

Now we shall describe how to construct, for a given CNF (an instance of 3SAT), an instance of the P/A graph assignment problem. Consider e.g. a clause $(a \vee b \vee \neg c)$. There exist seven mutually exclusive assignments of truth values to variables *a*, *b*, and *c* satisfying this clause (each assignment except of *a = false*, *b = false*, *c = true* is a satisfying one). We can model this clause using a "clause subgraph" which consists of a node for the clause, seven nodes for the mutually exclusive satisfying assignments, and six nodes for the values of propositional variables (three for positive values and three for negative values, that is, one for each literal). The clause node is connected to all assignment nodes by a fan-out alternative subgraph and each value node is connected to appropriate assignment nodes (those assignment nodes containing the literal which corresponds to the given value node) by a fan-in alternative subgraph. The following figure shows the clause subgraph for the clause $(a \vee b \vee \neg c)$, where capital letters in the assignment nodes represent value false (so for example aBc corresponds to *a = true*, *b = false*, *c = true*).



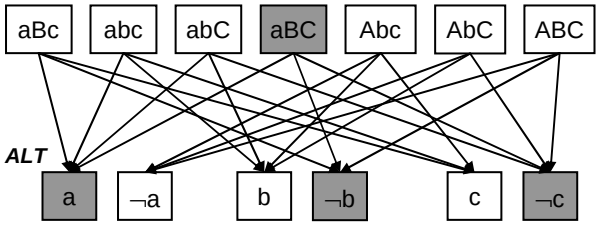
Each clause from the input CNF will be modelled using a clause graph with the above-described structure. To connect the clause graphs, we introduce a formula node and connect it with all clause nodes by a fan-out parallel subgraph. The formula node is forced to take value 1 (because we need the formula to be satisfied). A variable which is used in more than one clause will have value nodes in all clause graphs

where it appears. To interconnect these value nodes we introduce for each variable in the formula a variable node, which is forced to take value 1, and two literal nodes connected to the variable node by a fan-in alternative subgraph. Finally, each literal node is connected by a fan-in parallel subgraph to all value nodes in clause graphs which correspond to the given literal. The following figure shows these additional nodes and connections (the shaded nodes are the nodes that are forced to take value 1).



First let us observe that the number of nodes in the constructed P/A graph is linear in the size of the input CNF formula. Namely, if there are M clauses and N variables (and hence L = 3M literals) in the input CNF, then we get a graph with (14M + 3N + 1) nodes. Because 14M + 1 ≤ 5L and N ≤ L (assuming each variable appears at least once in the formula) we get that there are at most 8L nodes in the constructed P/A graph.

Now let us assume that the input CNF has a satisfying assignment. We shall construct a feasible assignment of the constructed P/A graph as follows. All clause nodes will get value 1 to satisfy the parallel fan-out from the formula node. The literal nodes of each variable will get the 0 and 1 values as defined by the satisfying assignment of the input CNF (for example if variable *b* is false in the satisfying assignment, then the node *b* gets the value 0 and the node ¬*b* gets the value 1). This satisfies the alternative fan-in into the variable nodes, and moreover it defines the 0 and 1 values for all value nodes via the parallel fan-ins that replicate the literal values into all clause subgraphs. Finally, for each clause exactly one assignment node is made valid, namely the one in which all three literals are valid, which satisfies the alternative fan-out from the clause nodes. It remains to show that also all alternative fan-ins into value nodes are satisfied. So let us consider an arbitrary value node. If it corresponds to a valid literal then it is connected to exactly one valid assignment node (the one where also the other two literals are valid), and if it corresponds to an invalid literal than it is connected only to invalid assignment nodes (see figure below). In both cases this is exactly what we need and hence the constructed assignment of 0/1 values to all nodes is feasible.



To complete the proof let us assume that there exists a feasible assignment of 0/1 values to all nodes of the constructed P/A graph. In this assignment:

- All clause nodes have value 1 to satisfy the parallel fan-out from the formula node.
- For each clause exactly one assignment node has value 1 to satisfy the alternative fan-out from the clause nodes.
- For each variable one literal node has value 1 and the other has value 0 to satisfy the alternative fan-ins into the variable nodes. The literal values are replicated into the value nodes by the parallel fan-ins into the literal nodes.

Now let us check that the truth assignment defined by the values assigned to the literal nodes satisfies the input CNF. To this end let us pick an arbitrary clause and assume by contradiction that it is falsified. That means that the three valid value nodes correspond to the only missing combination among the assignment nodes, or in other words that the only valid assignment node must be connected to an invalid value node. However, this is a contradiction, because the corresponding fan-in subgraph into this value node spoils the feasibility of the assignment (the principal node is 0 while one of its branching nodes is 1). Hence, the input CNF has a satisfying assignment if and only if the corresponding P/A graph has a feasible assignment. □

III. P/A Graph Pre-processing

Because solving the P/A graph assignment problem is hard, we focus now on inferring some information from the graph that can be used later to improve solver efficiency. In particular, we describe a heuristic algorithm for finding equivalent nodes in the P/A graph. We call a set of nodes of the P/A graph equivalent if and only if the nodes are assigned the same value in all feasible assignments of 0/1 values to nodes. Unfortunately, the problem of finding the largest possible sets of equivalent nodes is also hard. Let us now formalize the problem and prove its hardness.

Definition 3: Let G be a P/A graph, S_1 be a set of nodes of G which are fixed to value 1, and S_0 be a set of nodes of G which are fixed to value 0. Let *u* and *v* be arbitrary two

nodes of G. Then *u* and *v* are called *equivalent* with respect to the partial assignment given by S_1 and S_0 if and only if there is no feasible assignment of G extending the partial assignment in which *u* and *v* are assigned different values.

Definition 4: The *P/A graph equivalence class problem* can be stated as follows: given a P/A graph G, a set S_1 of nodes of G which are assigned value 1, and a set S_0 of nodes of G which are assigned value 0, output the partition of all nodes of G into equivalence classes, that is, into such sets that every two nodes from the same set are equivalent and no two nodes from distinct sets are equivalent.

Proposition 2: The P/A graph equivalence class problem is NP-hard.

Proof: It is enough to prove the following: if there exists an algorithm A which solves the P/A graph equivalence class problem in polynomial time, then such an algorithm can be used to solve the P/A graph assignment problem (which is, as we have proved, NP-complete) in polynomial time. So let us assume that G is a P/A graph, S_1 is a nonempty set of nodes of G which are fixed to value 1, and the question is whether there exists a feasible assignment of values to the remaining vertices.

Before proceeding further let us first decide, what output do we expect from algorithm A in the case when no feasible assignment exists. Strictly speaking, in that case the algorithm should output a single equivalence class, as every two vertices are equivalent according to the definition (there exists no feasible assignment in which they are assigned different values). If we adhere to this strict interpretation, a single run of A solves the assignment problem. Indeed, if all vertices end up in a single class then it suffices to test whether the all-one assignment is feasible (feasible assignment exists) or not (feasible assignment does not exist). If algorithm A returns at least two equivalence classes then a feasible assignment exists. However, it may be reasonable to look at the requirements on A in the following weaker way: if no feasible assignment exists then there is an empty set of restrictions on equivalence classes and A may output arbitrary sets. So let us now consider this less strict version of A. After running A, three situations may happen:

- The set S_1 is split among several (at least two) equivalence classes. Recall, that vertices in S_1 are pre-assigned to 1 so they are equivalent. If algorithm A puts them in different equivalence classes then it corresponds to non-existence of feasible assignment as described above (algorithm A outputs arbitrary sets).
- All vertices end up in a single class. As above, in this case it suffices to test whether the all-one assignment is feasible (feasible assignment exists) or not (feasible assignment does not exist).
- The set S_1 is contained in a single equivalence class (let us denote it by C) but there exist at least one additional nonempty equivalence class.

In the last case we shall proceed as follows. Let *u* be an arbitrary node not in C, that is, *u* is not equivalent with nodes in S_1. If there exists a feasible assignment, then there exists at least one in which *u* gets value 0 (otherwise *u* is equivalent with nodes in S_1). So we may proceed by setting $S_1 \leftarrow C$ and $S_0 \leftarrow S_0 \cup \{u\}$, and running A on the new data. Again, the above three cases may happen, and we iterate the process, until either it terminates by arriving to the first or second case, or it subsequently inserts all nodes in $S_0 \cup S_1$ (in which case there is a single assignment to test for feasibility). Since every run of A adds at least one node to the set $S_0 \cup S_1$, the maximum number of runs of A is bounded by the number of nodes in G. Therefore, if A runs in polynomial time, then so does the above described algorithm for testing the existence of a feasible assignment. Notice that the algorithm not only decides about the existence of feasible assignment, but in fact it constructs a feasible assignment if one exists. □

Since the problem of P/A graph equivalence class is hard, we focus only on discovering certain typical situations at this stage. The most important situation we want to recognize consists of a process which splits in several sub-processes in alternative branching and all these sub-processes join afterwards. Principal nodes where the production process splits and sub-processes join back again are equivalent. We are looking for the algorithm that can discover at least such equivalence classes.

The proposed algorithm has two phases. In the initial phase an undirected hyper-graph is constructed from the input P/A graph. The constructed hyper-graph has almost the same structure and represents almost the same information about the production processes as the original P/A graph. Only the directions of arcs and hence precedence relations are omitted, which is not a problem because the input P/A graph is acyclic so the precedence relations trivially hold (actually, the precedence relations are used only to define fan-in and fan-out subgraphs in acyclic P/A graphs).

The second and major phase of the algorithm repeatedly transforms the given hyper-graph using certain transformation rules into a simpler and more explicit hyper-graph. Sets of equivalent nodes of the input P/A graph are built along these transformation steps. This phase terminates when no transformation rule can be applied or when a conflict in the hyper-graph is detected.

A. Initial Phase of the Algorithm

Let G = (V, E) together with a set of marked fan-in and fan-out sub-graphs be a P/A graph (Figure 2). Some nodes of G may be forced to take value 1 (filled by gray in Figure 2) and some nodes may be fixed to value 0. We construct a hyper-graph H = (U, F) over a set of nodes U obtained by adding two extra nodes 0 and 1 to the original set V. The extra nodes 0 and 1 represent equivalence classes of nodes which are always 0 and 1 respectively in all feasible assignments. The construction of the set of hyper-edges F is done according to the input graph G in the following way.

Let set F be an empty set at the beginning.

- For each *fan-in parallel* sub-graph of G over nodes x, y_1, y_2,…, y_k, where x is the principal node, insert edges $\{\{x\}, \{y_i\}\}$ for $1 \le i \le k$ into F.
- *Fan-out* sub-graphs of G marked as *parallel* are treated in the same way as fan-in parallel sub-graphs.
- For each *fan-in alternative* sub-graph of G over nodes x, y_1, y_2,…, y_k, where x is the principal node, insert non-trivial hyper-edge $\{\{x\}, \{y_1, y_2,\ldots, y_k\}\}$ into F.
- As in the case of parallel branching, *fan-out* sub-graphs of G marked as *alternative* are treated in the same way as fan-in alternative sub-graphs.
- For each node x of G which is *forced* to take value 1 we insert edge $\{\{x\}, \{1\}\}$ into F. An analogical edge addition is done for nodes which are fixed to value 0. Edge $\{\{y\}, \{0\}\}$ is inserted into F for each node y of G which is forced to take value 0.

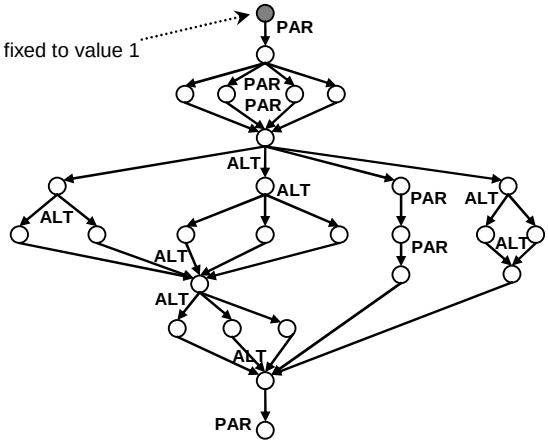


Fig. 2. Example of P/A graph with PAR/ALT annotation.

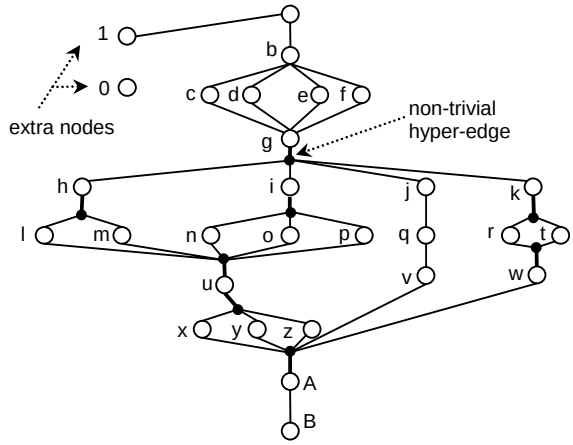


Fig. 3. Hyper-graph corresponding to the P/A graph.

Informally speaking, we use the same nodes in the hyper-graph as in the P/A graph. For each arc that is a part of parallel branching in the P/A graph we add an edge between the same nodes in the hyper-graph. For a set of arcs that form alternative branching in the P/A graph, we add a non-trivial hyper-edge connecting the same nodes in the hyper-graph (a small black dot in Figure 3). We use the convention that an edge with the same structure is added only once (we do not allow multi-edges). Figure 3 shows a hyper-graph constructed for the P/A graph from Figure 2.

The last step of the initial phase is constructing the initial equivalence classes. Let us denote Q_u the equivalence class for $u \in U$. Initially we set $Q_u = \{u\}$ for every $u \in U$. The constructed hyper-graph $H = (U, F)$ with associated equivalence classes is used as the input for the transformation phase of the algorithm.

B. Transformation Phase of the Algorithm

The goal of the transformation phase is to modify the hyper-graph while preserving the equivalence classes. This is realized by several transformation rules that update the hyper-graph by adding derived hyper-arcs. During these updates, the initial equivalence classes are being merged.

Edge contraction rule. The first transformation rule contracts an edge. An edge $\{\{u\}, \{v\}\} \in F$ can be contracted if for any non-trivial hyper-edge $\{\{x\},Y\} \in F$ $| \{u,v\} \cap (\{x\} \cup Y) | \le 1$ and $| \{0,1\} \cap (Q_u \cup Q_v) | \le 1$. The case when an edge cannot be contracted is treated separately (see the rules below). The edge contraction rule represents a standard operation from the graph theory.

Let $\{\{u\}, \{v\}\} \in F$ be the edge that can be contracted. Then the following steps are carried out. Erase vertex v by assigning: $U \leftarrow U - \{v\}$ and $Q_u \leftarrow Q_u \cup Q_v$. Edges and hyper-edges that contain v need to be modified to form a correct hyper-graph without v. If there is an edge $\{\{x\},\{v\}\} \in F$, where $x \ne u$, replace it by edge $\{\{x\},\{u\}\}$. If there is a non-trivial hyper-edge $\{\{v\},Y\} \in F$ then replace it by $\{\{u\},Y\}$. If there is a non-trivial hyper-edge $\{\{x\},Y\} \in F$, where $v \in Y$, $u \notin Y$, and $x \ne u$ then replace it by hyper-edge $\{\{x\}, (Y - \{v\}) \cup \{u\}\}$.

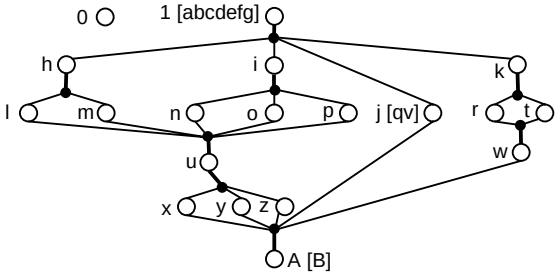


Fig. 4. Hyper-graph after edge contractions.

Figure 4 shows a hyper-graph after applying the edge contraction rule to the hyper-graph in Figure 3. Namely, we contracted all edges between nodes *a* and *g* (and 1) to obtain a single node for equivalence class {1,*a*,*b*,*c*,*d*,*e*,*f*,*g*}. We also contracted edges $\{\{j\}, \{q\}\}$ and $\{\{q\}, \{v\}\}$ and got a node with equivalence class $\{j,q,v\}$. Finally, we contracted edge $\{\{A\}, \{B\}\}$ to obtain a node for equivalence class $\{A,B\}$.

Hyper-edge extension rule. If there are non-trivial hyper-edges $\{\{x\}, Y\} \in F$ and $\{\{y\}, Z\} \in F$, where $y \in Y$ and $(\{x\} \cup Y) \cap (\{y\} \cup Z) = \{y\}$ then add a new hyper-edge $\{\{x\}, Y \cup Z - \{y\}\}$ into F.

Hyper-edge meet rule. If there are non-trivial hyper-edges $\{\{x\},Y\} \in F$ and $\{Z, \{w\}\} \in F$, where $x \neq w$ and $Z \subseteq Y$ then add a new hyper-edge $\{\{x\}, (Y–Z) \cup \{w\}\}$ into F. Again the rule has no effect if the edge generated by this rule is already present in the hyper-graph. A special case of this rule when Y = Z results in addition of a new edge $\{\{x\}, \{w\}\}$ that can be contracted subsequently. This transformation rule in cooperation with the previous rule can discover the situation when a production chain splits into several alternatives and all these alternatives join again.

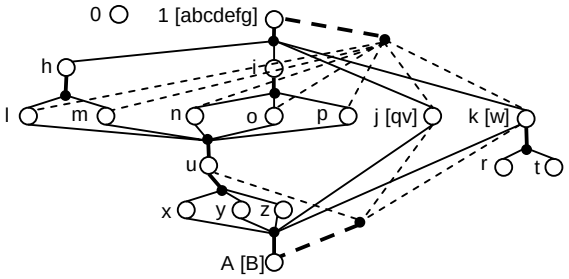


Fig. 5. Hyper-graph after extension and meet rule application.

The hyper-graph in Figure 5 was obtained by applying the hyper-edge meet rule to hyper edges $\{\{k\}, \{r,t\}\}$ and $\{\{w\}, \{r,t\}\}$ that lead to adding a new edge $\{\{k\}, \{w\}\}$. This edge was then contracted to find a new equivalence class $\{k,w\}$. The new bottom hyper-edge $\{\{A\}, \{u,j,k\}\}$ was then obtained by applying the hyper-edge meet rule using hyper-edge $\{\{u\}, \{x,y,z\}\}$. The new top hyper-edge was obtained by applying hyper-edge extension rule.

Never-valid activity detection rule (case A). If there is a non-trivial hyper-edge $\{\{x\},Y\} \in F$ and an edge $\{\{y_1\}, \{y_2\}\} \in F$, where $\{y_1, y_2\} \subseteq Y$ (notice that the edge $\{\{y_1\}, \{y_2\}\}$ cannot be contracted in this situation since contraction precondition is not satisfied), then replace the hyper-edge $\{\{x\},Y\}$ by a new hyper-edge $\{\{x\}, Y–\{y_1,y_2\}\}$ and add new edges $\{\{0\}, \{y_1\}\}$ and $\{\{0\}, \{y_2\}\}$.

Never-valid activity detection rule (case B). If there is a non-trivial hyper-edge $\{\{x\},Y\} \in F$ and an edge $\{\{x\}, \{y\}\} \in F$, where $y \in Y$ (again notice that the edge $\{\{x\}, \{y\}\}$ cannot be contracted in this situation), then erase the hyper-edge $\{\{x\},Y\}$ from F and add new edges $\{\{0\}, \{z\}\} \in F$ for all $z \in Y - \{y\}$.

Conflict detection rule. The case when an edge $\{\{u\}, \{v\}\}$ cannot be contracted due to violated inequality $|\{0,1\} \cap (Q_u \cup Q_v)| \leq 1$ (i.e. $|\{0,1\} \cap (Q_u \cup Q_v)| = 2$ holds) indicates a conflict. This conflict arises because two equivalent nodes *u* and *v* are enforced to different values and hence no feasible assignment for the given input exists.

If any of the above-defined transformation rules cannot be applied, the algorithm terminates with success. After successful termination the node equivalence classes associated with nodes remaining in the final hyper-graph represent sets of equivalent nodes.

C. Correctness of the algorithm

We shall show now that the algorithm is correct, that is, it returns groups of equivalent nodes. We can easily define feasible assignment of 0/1 values to nodes of the hyper-graph. This definition should ensure that an assignment of 0/1 values to nodes of the original P/A graph is feasible if and only if it is feasible for the corresponding hyper-graph (the correspondence between the P/A graph and the hyper-graph is given by the initial phase of the algorithm).

Let us denote $Q = \cup_{v \in V} Q_v$. An assignment $val: Q \rightarrow \{0,1\}$ in hyper-graph $H = (U, F)$ is feasible if and only if $val(0) = 0$, $val(1) = 1$, for every edge $\{\{u\}, \{v\}\} \in F$ $val(u) = val(v)$, and for every non-trivial hyper-edge $\{\{x\},Y\} \in F$ $\Sigma_{y \in Y} val(y) = val(x)$. Finally, the nodes in the same equivalence class associated with a given node have assigned the same value.

To prove the correctness of the algorithm it is sufficient to show that hyper-graph transformation rules preserve feasible assignments. The proofs are based on equivalence of arithmetic expressions describing the branching before and after applying the particular transformation rule.

Proposition 3 (correctness of edge contraction). An assignment $val: Q \rightarrow \{0,1\}$ in the hyper-graph $H = (U,F)$ is feasible if and only if it is feasible for the hyper-graph after application of the *edge contraction rule*.

Proof. Let $\{\{u\}, \{v\}\} \in F$ be the contracted edge and $\{\{x\}, \{v\}\} \in F$, where $x \neq u$, be another edge (if any). Then feasibility for hyper-graph H enforces $val(x) = val(v)$ and $val(u) = val(v)$. Feasibility for hyper-graph after edge contraction enforces $val(x) = val(u)$ since $\{\{x\}, \{v\}\}$ is replaced by $\{\{x\}, \{u\}\}$ and $val(u) = val(v)$ since $v \in Q_u$. These two sets of formulas are clearly equivalent.

For hyper-edge $\{\{v\},Y\} \in F$ feasibility for H enforces $\Sigma_{y \in Y} val(y) = val(v)$ and $val(u) = val(v)$. Feasibility for hyper-graph after edge contraction enforces $\Sigma_{y \in Y} val(y) = val(u)$ since $\{\{v\},Y\}$ is replaced by $\{\{u\},Y\}$ and $val(u) = val(v)$ since $v \in Q_u$. These two sets of formulas are equivalent.

Finally, for hyper-edge $\{\{x\},Y\} \in F$, where $u \notin Y$, $v \in Y$ and $x \neq u$, feasibility for H enforces $\Sigma_{y \in Y} val(y) = val(x)$ and $val(u) = val(v)$. Feasibility for hyper-graph after edge contraction enforces $\Sigma_{y \in Y} val(y) – val(v) + val(u) = val(x)$ and $val(u) = val(v)$ since $v \in Q_u$. These two sets of formulas are again equivalent. □

Proposition 4 (correctness of hyper-edge extension). An assignment $val: Q \rightarrow \{0,1\}$ in the hyper-graph $H = (U, F)$ is feasible if and only if it is feasible for the hyper-graph after application of the *hyper-edge extension rule*.

Proof. Let $\{\{x\},Y\} \in F$ and $\{\{y\},Z\} \in F$, where $y \in Y$ and $(\{x\} \cup Y) \cap (\{y\} \cup Z) = \{y\}$, be non-trivial hyper-edges that are selected to form a new hyper-edge. Feasibility for hyper-graph H enforces $\Sigma_{z \in Y} val(z) = val(x)$, $\Sigma_{z \in Z} val(z) = val(y)$, which implies $\Sigma_{z \in Y} val(z) + \Sigma_{z \in Z} val(z) - val(y) = val(x)$. This is exactly the constraint enforced by the new hyper-edge. □

Proposition 5 (correctness of hyper-edge meet). An assignment $val: Q \rightarrow \{0,1\}$ in the hyper-graph H = (U,F) is feasible if and only if it is feasible for the hyper-graph after application of the *hyper-edge meet rule*.

Proof. Let $\{\{x\},Y\} \in F$ and $\{Z, \{w\}\} \in F$, where $x \neq w$ and $Z \subseteq Y$, be non-trivial hyper-edges on which the rule is applied. Feasibility for hyper-graph H enforces $\Sigma_{y \in Y} val(y) = val(x)$ and $\Sigma_{z \in Z} val(z) = val(w)$, which implies $\Sigma_{y \in Y} val(y) - \Sigma_{z \in Z} val(z) + val(w) = val(x)$. This is exactly the constraint enforced by the added hyper-edge. □

Proposition 6 (correctness of never-valid activity detection). An assignment $val: Q \rightarrow \{0,1\}$ in the hyper-graph H = (U, F) is feasible if and only if it is feasible for the hyper-graph after application of the *never-valid activity detection rule*.

Proof. Let us prove the case A of the rule first. Let $\{\{x\},Y\} \in F$ and $\{\{y_1\},\{y_2\}\} \in F$, where $\{y_1,y_2\} \subseteq Y$ be (hyper-)edges in H. The feasible assignment *val* in H must satisfy $val(y_1) = val(y_2) = 0$, since the remaining combinations of assignments of 0/1 values to y_1 and y_2 would violate the constraint $\Sigma_{y \in Y} val(y) = val(x)$. Hence, for any feasible assignment the constraint $\Sigma_{y \in Y} val(y) = val(x)$ holds if and only if $\Sigma_{y \in Y - \{y_1,y_2\}} val(y) = val(x)$ holds. Thus we did not change the set of feasible assignments for the hyper-graph by replacing the hyper-edge $\{\{x\},Y\}$ by $\{\{x\},Y-\{y_1,y_2\}\}$ and by adding new edges $\{\{0\},\{y_1\}\}$ and $\{\{0\},\{y_2\}\}$ that imply $val(y_1) = 0$ and $val(y_2) = 0$.

The proof of case B of the rule is similar. Let $\{\{x\},Y\} \in F$ and $\{\{x\},\{y\}\} \in F$, where $y \in Y$, be (hyper-)edges in H. Clearly, every feasible assignment *val* must satisfy $val(x) = val(y)$ and for all $z \in Y-\{y\}$ $val(z) = 0$. Any other assignment would violate the constraint $\Sigma_{y \in X} val(y) = val(x)$. Hence, addition of edges $\{\{0\},\{z\}\}$ for all $z \in Y-\{y\}$ to F does not change the set of feasible assignments and the constraint on feasible assignments induced by the removed hyper-edge $\{\{x\},Y\}$ is subsumed by constraints induced by newly added edges and by the edge $\{\{x\},\{y\}\}$. □

IV. Temporal Networks with Alternatives

So far we assumed acyclic graphs describing precedence relations between nodes and we focused on the logical aspects of the network, namely selecting the nodes to satisfy parallel and alternative branching. Nevertheless, in real-life problems we usually need a finer time resolution so we can extend precedence relations to simple temporal relations. In particular, each arc (X,Y) in the P/A graph 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, $a \leq Y - X \leq b$. We call the resulting graph a *P/A simple temporal network*. Now the problem is to decide validity of nodes satisfying parallel and alternative branching and to assign time (number) to each valid node in such a way that all simple temporal relations between the valid nodes are satisfied. We call the problem of deciding whether a feasible assignment of validity and time variables exists a *P/A simple temporal network assignment problem*. Again, we assume that validity of some nodes is set to 1 (otherwise, there is a trivial solution where all nodes are invalid). This is a typical situation when the proposed temporal network is used to model real-life problems. The last nodes in the structure of alternative process routes typically describe deliveries to customers. Because the deliveries must be fulfilled and we can just select alternative ways how to do it, these nodes must be valid.

Recall, that there exist polynomial algorithms for checking consistency of simple temporal networks [4] so solving simple temporal problems is "easy". However, as we showed above adding parallel and alternative branching makes the problem hard.

Proposition 7: The P/A simple temporal network assignment problem is NP-complete.

Proof: The proof is straightforward if we realise that P/A graphs are just special cases of P/A simple temporal networks. In particular, for any P/A graph we can construct a P/A simple temporal network where all temporal constraints are in the form $[0,\infty)$. Now, there exists a feasible assignment to the P/A graph if and only if there exists a feasible assignment to the corresponding P/A simple temporal network (all time variables can be set to 0, which trivially satisfies all temporal constraints). □

It may seem that we can further generalise the framework by using a disjunction of simple temporal relations that are used in Temporal Constraint Satisfaction Problems [4]. However, this generalisation does not increase the expressive power of the framework because temporal disjunctions in the form $\vee_{i=1,..,n} a_i \leq Y - X \leq b_i$ can be substituted by a sub-network with simple temporal constraints as Figure 6 shows. Note that auxiliary nodes x' and y' are necessary to keep fan-out subgraph with principal node X or fan-in subgraph with principal node Y. Nodes x' and y' are equivalent in the sense described in the previous section and the algorithm presented there can detect this equivalence.

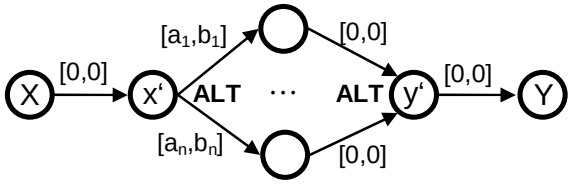


Fig. 6. Modelling simple disjunctions of temporal constraints.

V. Related Works

The intended application area for the proposed framework is manufacturing scheduling. There exists a benchmark set MaScLib by ILOG [9] which contains a formal description of real-life manufacturing scheduling problems. This description includes the concept of validity variables and logical dependencies between them. Temporal and logical relations are modelled separately there and various binary logical relations can be defined between the validity variables. Our framework defines the logical dependencies via branching in the temporal graph. According to our experience this is satisfactory for modelling manufacturing (and other) processes. Moreover, we believe that the coupled definition will lead to more efficient filtering algorithms that use together temporal and logical information. In [2] we already showed that integrated filtering of precedence and dependency constraints significantly reduces solving time.

We are not aware about another approach that can handle alternative process routes in the same generality as the proposed P/A simple temporal networks. The paper [5] describes a graph concept for modelling alternative processes, but it cannot be used for alternative routes because all activities must be present. Probably the closest approach to our proposal is the work by Beck and Fox [3] on modelling alternative processes using PEX (probability of existence) variables. In our framework we focus on logical validity variables (PEX uses an interval of real numbers $\langle 0,1 \rangle$) but the main ideas of propagation are very similar. Using validity variables instead of PEX values simplifies integration to existing constraint solvers and we believe that using logical deduction during pre-processing can generate additional input to the filtering algorithm.

Our work is naturally related to temporal networks as we proposed an extension of simple temporal networks. We already showed that the proposed framework covers Temporal Constraint Satisfaction Problems [4]. Disjunctive Temporal Network [10] is another approach to handling temporal alternatives. We have no formal comparison to our P/A simple temporal network yet, but our ambition is slightly different from DTN – we model alternative routes rather than any temporal disjunction.

Several other extensions of temporal networks appeared such as resource temporal networks [7] or disjunctive temporal networks with finite domain constraints [8]. These extensions integrate temporal reasoning with reasoning on non-temporal information, such as fluent resources. Our ambition is to extend existing constraint-based scheduling by some planning decisions, namely selection of alternative processes. So we extended temporal reasoning by logical reasoning on existence of nodes in the network. Actually, the possibility to decide about validity/invalidity of the node is the main difference of our approach from the above mentioned works on temporal networks where all nodes must always be present. Moreover, we also plan to include resource reasoning in our framework, namely including disjunctive resource constraints. The paper [1] presents such a constraint that can handle activities with the validity status.

There exists Conditional Temporal Planning [11] where existence of node in the network depends on a certain condition. Though there is some similarity in modelling alternative processes/plans, satisfaction of condition in CTP depends on external forces – Nature – rather than being an internal relation between the nodes. In our approach, decision of validity of the node is done internally based on logical relations between the nodes.

VI. Conclusions

The paper reports a work in progress on extension of simple temporal networks towards handling alternative process routes. We focused on formalizing this new modelling framework, showing its complexity, and proposing a pre-processing step for extracting information about logically equivalent nodes in the network.

The proposed framework combined with existing resource constraints is aimed at solving complex manufacturing scheduling problems where resource and time allocation interleaves with selection of best processes to satisfy customer demands. Hence, we contribute to the area of integrated planning and scheduling techniques by extending traditional scheduling technology by formal reasoning on alternative plans/processes.

References

[1] Barták, R., "Incremental Propagation of Time Windows on Disjunctive Resources," *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference*, AAAI Press, pp. 25-30, 2006.

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

[3] Beck, J.Ch. and Fox, M.S., "Scheduling Alternative Activities," *Proceedings of AAAI-99*, USA, pp. 680-687, 1999.

[4] Dechter, R.; Meiri, I. and Pearl, J., "Temporal Constraint Networks," *Artificial Intelligence*, 49:61.95, 1991.

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

[6] Garey, M. R. and Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, 1979.

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

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

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

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

[11] Tsamardinos, I.; Vidal, T. and Pollack, M.E., "CTP: A New Constraint-Based Formalism for Conditional Temporal Planning," *Constraints*, 8(4):365.388, 2003.