A Constraint Model for State Transitions in Disjunctive Resources

Roman Barták*, Ondřej Čepek*†

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

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

Abstract. Traditional resources in scheduling are simple machines where a capacity is the main restriction. However, in practice there frequently appear resources with more complex behaviour that is described using state transition diagrams. This paper presents new filtering rules for constraints modelling the state transition diagrams. These rules are based on the idea of extending traditional precedence graphs by direct precedence relations. The proposed model also assumes optional activities and it can be used as an open model accepting new activities during the solving process.

1 Introduction

Temporal networks play an important role in planning but they are not used as frequently in scheduling where resource restrictions traditionally play a stronger role. This is reflected in scheduling global constraints, where techniques like edge-finding or not-first/not-last combine restrictions on time windows with a limited capacity of the resource [1]. Recently, a new category of propagation techniques combining information about relative position of activities with capacity of resources appeared [4]. Also techniques combining information about precedence relations and time windows have been proposed [8]. We believe that integration of temporal networks with reasoning on resources [9,10] will play even more important role as planning and scheduling technologies are becoming closer.

In this paper we propose an extension of precedence graphs by direct precedence relations (A can directly precede B if no activity must be allocated between A and B). This extension is motivated by modelling complex behaviour of resources that is described as a state transition diagram. Such a diagram is in fact a generalisation of set-up times that play an important role in current real-life scheduling problems. As factories are transforming from mass production to more customised production, multi-purpose and hence more complicated machines are used and better handling of setups is becoming important. Also over-subscribed problems are more frequent nowadays so the scheduling systems should be able to handle optional activities, for example, to decide about rejection of activity that cannot be scheduled feasibly to-

gether with other activities. Note also, that optional activities are useful for modelling alternative resources (an optional activity is used for each alternative resource) as well as alternative processes to accomplish a job (each process may consist of one of several different sets of activities). Scheduling systems should also be able to add activities to satisfy the transition scheme, for example to insert a setup activity if necessary.

To summarise the contributions of this paper, we propose two extensions of ordinary precedence graphs: adding direct precedence relations and using optional activities. For such a graph which we call a *double precedence graph* we design incremental filtering rules that keep a transitive closure of the graph and deduce new precedences and (in)validity of activities. Moreover, opposite to traditional global constraints used in scheduling the proposed model is open, that is, it allows adding new activities to the precedence graph during the solution process.

2 Motivation

In this paper we address the problem of modelling a *disjunctive resource* where activities must be allocated in such a way that they do not overlap in time. We assume that there are precedence constraints between the activities. The precedence constraint A « B specifies that activity A must be before activity B in the schedule. Each activity is annotated by a resource state requested for processing the activity and there is a state transition diagram describing transitions between the states. *State transition diagram* is a directed graph where nodes describe the states and arcs describe allowed transitions between the states (Figure 1). The state transition diagram restricts sequencing of activities in the following way: activity A can be scheduled directly before activity B only if there is an arc from the state of A to the state of B in the state transition diagram. To model over-subscribed problems and alternative resources/processes, we assume *optional activities*. An optional activity has one of the following three statuses. If the activity is not yet known to be or not to be included then it is called *undecided*. If the activity is allocated to the resource then it is called *valid*. If the activity is known not to be allocated to the resource then it is called *invalid*. Regular activities correspond to valid activities. The scheduling task is to decide about (in)validity of the undecided activities and to find a sequence of valid activities satisfying the precedence constraints and restrictions imposed by the state transition diagram.

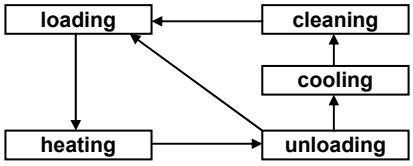

Fig. 1. Example of a state transition diagram

In real-life problems there are usually also time windows restricting position of activity in time. In such a case, it is known that deciding about an existence of a feasible schedule is NP-hard in the strong sense [7] so there is a little hope even for a pseudo-

polynomial solving algorithm. Hence using propagation rules and constraint satisfaction techniques is justified here. The paper [2] shows how filtering of time windows can be combined with the precedence graph so in this paper we focus merely on handling (direct) precedence relations. Our goal is to propose filtering rules that remove inconsistencies from the double precedence graph.

3 Related Works

Disjunctive temporal networks [11] can model disjunctive resources. However, DTNs use more general disjunctions than necessary and hence they achieve weaker pruning. Moreover, a qualitative approach to time seems more appropriate to describe the problem of activity sequencing. Though we assume durative activities, interval algebra is superfluous because disjunctive resources discard most of the interval relations (like starts, during, overlaps etc.). From Point Algebra we need only 'before' and 'after' relations and there is no support for direct precedences there. The work by Laborie [9] studies a combination of resource and temporal reasoning but no algorithm is presented (and a different type of resources is assumed). Probably the closest approach to our problem is presented in the paper [6] where alternative resources correspond to paths in the global precedence graph. However, this approach is proposed merely for cost-based filtering (optimization of makespan or setup times) and it assumes all the activities to be present in the global precedence graph. The paper [3] presents an idea of a precedence graph with optional activities. The authors use a so called PEX value to describe a probability of the existence of the activity and their approach is based on updating this value. Instead of that we use a Boolean variable to describe the presence of an activity and we focus more on precedence and direct precedence relations. To summarise the above discussion, none of the existing approaches to temporal and resource reasoning covers fully state transition diagrams and optional activities.

3 Double Precedence Graphs

The precedence relations among activities define a *precedence graph* that is an acyclic directed graph where nodes correspond to activities and there is an arc from A to B if A « B. If access to all predecessors and successors of a given activity is frequently requested, like in [4,8], then it is more efficient to keep a transitive closure of the graph where this information is available in time O(1) rather than to look for predecessors/successors on demand. We propose the following definition of transitive closure of the precedence graph with optional activities.

Definition 1: We say that a precedence graph G with optional activities is *transitively closed* if for any two arcs A to B and B to C such that B is a valid activity and A and C are either valid or undecided activities there is also an arc A to C in G.

It is easy to prove that if there is a path from A to B such that A and B are either valid or undecided and all inner nodes in the path are valid then there is also an arc from A to B in a transitively closed graph (by induction on the path length). Hence, if no

optional activity is used (all activities are valid) then Definition 1 corresponds to a standard definition of the transitive closure.

To model restrictions imposed by the state transition diagram we propose to extend the precedence graph by direct precedence relations between the activities.

Definition 2: We say that A can *directly precede* B if both A and B are either valid or undecided activities, B is not before A ($\neg$ B « A), the transition from A to B is allowed by the state transition diagram, and there is no valid activity C such that A « C and C « B (the relation « is from the transitive closure of the precedence graph with optional activities).

The relation of direct precedence introduces a new type of arc, say «$_d$, in the precedence graph and hence we are speaking about the *double precedence graph*. There is one significant difference between the arcs of type « and the arcs of type «$_d$. While the arcs « are added into the graph as problem solving proceeds, the arcs «$_d$ are typically removed from the graph (note that «$_d$ means "can be directly before", while « means "must be before"). When all valid activities are linearly ordered, there is exactly one arc of type «$_d$ going into each valid activity (with the exception of the very first activity in the schedule) and one arc of type «$_d$ going from each valid activity (with the exception of the very last activity in the schedule).

Constraint Model

We propose to realise a reasoning on precedence relations using constraint satisfaction technology. This allows integration of our model with other constraint reasoning techniques [2]. This integration requires the model to provide full information about precedence relations to all other constraints. We index each activity by a unique number from the set 1,..,n, where n is the number of activities. For each activity we use a 0/1 variable Valid indicating whether the activity is valid (1) or invalid (0). If the activity is undecided – not yet known to be valid or invalid – then the domain of Valid is {0,1}. The precedence graph is encoded in two sets attached to each activity. CanBeBefore(A) is a set of indices of activities that can be before activity A. CanBeAfter(A) is a set of indices of activities that can be after activity A. For simplicity reasons we will write A instead of the index of A. To simplify description of the propagation rules we define for every activity A the following derived sets:

MustBeAfter(A) = CanBeAfter(A) \ CanBeBefore(A)
MustBeBefore(A) = CanBeBefore(A) \ CanBeAfter(A)
Unknown(A) = CanBeBefore(A) $\cap$ CanBeAfter(A).

MustBeAfter(A) and MustBeBefore(A) are sets of those activities that must be after and before the given activity A respectively. Unknown(A) is a set of activities that are not yet known to be before or after activity A (Figure 2).

To model direct precedence relations and hence a double precedence graph, we add two sets to each activity: CanBeRightBefore and CanBeRightAfter containing indexes of activities that can be directly before and directly after a given activity. Naturally, the following relation holds CanBeRightBefore(A) $\subseteq$ CanBeBefore(A) at any time and similarly CanBeRightAfter(A) $\subseteq$ CanBeAfter(A).

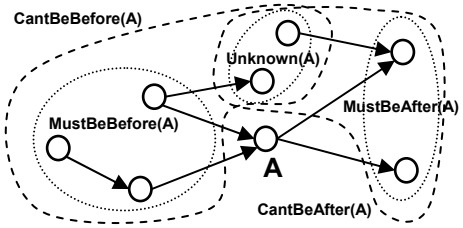

Fig. 2. Representation of the precedence graph

Note on representation. The main reason for using sets to model the precedence graph is their possible representation as domains of variables in constraint satisfaction packages. Recall that domains of variables can only shrink as problem solving proceeds. The sets in our model are also shrinking as new arcs « are added to the precedence graph. Hence a special data structure is not necessary to describe the precedence graph in constraint satisfaction packages. Moreover, these packages usually provide tools to manipulate the domains, for example membership and deletion operations. In the subsequent complexity analysis, we will assume that these operations require time O(1), which can be realised for example by using a bitmap representation of the sets. Note finally, that empty domain implies inconsistency that may be a problem for the very first and very last activity which has no predecessors and successors respectively. To solve the problem we can simply leave activity A in both sets CanBeAfter(A) and CanBeBefore(A). Then no domain of CanBeBefore and CanBeAfter will ever be empty but we can detect inconsistency via the empty domain of Valid variables.

Propagation Rules for Simple Precedences

The goal of propagation rules is to remove inconsistent elements (activities) from the above described sets – this is called domain filtering in constraint satisfaction. In the first stage, we will focus on making a transitive closure of the precedence graph according to Definition 1. Note that the transitive closure of the precedence graph also simplifies detection of inconsistency of the graph. The precedence graph is inconsistent if there is a cycle of valid activities. In a transitively closed graph, each such cycle can be detected by finding two valid activities such that A « B and B « A. Our propagation rules prevent cycles by making invalid the last undecided activity in each cycle. This propagation is realised by using an exclusion constraint. As soon as there is a cycle A « B and B « A detected, the following exclusion constraint can be posted:

$$\text{Valid}(A) = 0 \vee \text{Valid}(B) = 0.$$

This constraint ensures that each cycle is broken by making at least one activity in the cycle invalid. Instead of posting the constraint directly to the constraint solver, we propose keeping the set Ex of exclusions. The above exclusion constraint is modelled

as a set {A,B} ∈ Ex. Now, the propagation of exclusions is realised explicitly – if activity A becomes valid then all activities C such that {A,C} ∈ Ex are made invalid.

We initiate the precedence graph in the following way. First, the variables Valid(A), CanBeBefore(A), CanBeRightBefore(A), CanBeAfter(A), and CanBeRightAfter(A) with their domains are created for every activity A. Then the known precedence relations in the form A « B are added by removing B from the sets CanBeBefore(A) and CanBeRightBefore(A), and removing A from the sets CanBeAfter(B) and CanBeRightAfter(B). Note, that because all activities are still undecided at this stage, domain change is not propagated to other variables. Finally, the Valid(A) variable for every valid activity A is set to 1 (and similarly Valid variables of invalid activities are set to 0). By instantiating the Valid(A) variable, the propagation rule /1/ is invoked. "Valid(A) is instantiated" is its trigger. The part after → is a propagator describing pruning of domains. "exit" means that the constraint represented by the propagation rule is entailed so the propagator is not further invoked (its invocation does not cause further domain pruning). We will use the same notation in all rules. The propagation rule /1/ realises the above described exclusion constraints as well as adding new arcs according to Definition 1.

```
Valid(A) is instantiated →                                              /1/
   if Valid(A) = 0 then
      Ex := Ex \ {{A,X} | X is an activity}
      for each B do          // disconnect A from B
       CanBeBefore(B) ← CanBeBefore(B) \ {A}
       CanBeAfter(B) ← CanBeAfter(B) \ {A}
       CanBeRightBefore(B) ← CanBeRightBefore(B) \ {A}
       CanBeRightAfter(B) ← CanBeRightAfter(B) \ {A}
   else  // Valid(A)=1
      for each C s.t. {A,C}∈Ex do Valid(C) ← 0
      for each B∈MustBeBefore(A) s.t. Valid(B)≠0 do
       for each C∈MustBeAfter(A) s.t. Valid(C)≠0 do
         CanBeRightAfter(B) ← CanBeRightAfter(B) \ {C}
         CanBeRightBefore(C) ← CanBeRightBefore(C) \ {B}
         if C∉MustBeAfter(B) then
           CanBeAfter(C) ← CanBeAfter(C) \ {B} //add arc from B to C
           CanBeBefore(B) ← CanBeBefore(B) \ {C}
           CanBeRightAfter(C) ← CanBeRightAfter(C) \ {B}
           CanBeRightBefore(B) ← CanBeRightBefore(B) \ {C}
           if C∉CanBeAfter(B) then    // break the cycle
              if Valid(B)=1 then Valid(C) ← 0 // Valid(C)=1 leads to fail
              else if Valid(C)=1 then Valid(B) ← 0
                   else Ex ← Ex ∪ {{B,C}}
   exit
```

Note that rule /1/ maintains symmetry of sets modelling the double precedence graph for all valid and undecided activities because the domains are pruned symmetrically in pairs. We shall show now, that if the entire precedence graph is known in advance (no arcs are added during the solving procedure), then rule /1/ is sufficient for keeping the transitive closure according to Definition 1.

Proposition 1: Let A_0, A_1, … , A_m be a path in the precedence graph such that Valid(A_j)=1 for all $1 \le j \le m-1$ and Valid(A_0)≠0 and Valid(A_m)≠0 (that is, the endpoints of the path are not invalid and all inner points of the path are valid). Then A_0 « A_m, that is, $A_0 \notin$ CanBeAfter(A_m) and $A_m \notin$ CanBeBefore(A_0).

Proof: We shall proceed by induction on m. The base case m=1 is trivially true after initialisation (we assume that for every arc (X,Y) in the precedence graph X is removed from CanBeBefore(Y) and Y is removed from CanBeAfter(X) in the initialisation phase). For the induction step let us assume that the statement of the lemma holds for all paths (satisfying the assumptions of the lemma) of length at most m-1. Let $1 \le j \le m-1$ be an index such that $Valid(i_j) \leftarrow 1$ was set last among all inner points $i_1, \dots, i_{m-1}$ on the path. By the induction hypothesis we get

- $i_0 \notin CanBeAfter(i_j)$ and $i_j \notin CanBeBefore(i_0)$ using the path $i_0, \dots, i_j$
- $i_j \notin CanBeAfter(i_m)$ and $i_m \notin CanBeBefore(i_j)$ using the path $i_j, \dots, i_m$

We shall distinguish two cases. If $i_m \in MustBeAfter(i_0)$ (and thus by symmetry also $i_0 \in MustBeBefore(i_m)$) then by definition $i_m \notin CanBeBefore(i_0)$ and $i_0 \notin CanBeAfter(i_m)$ and so the claim is true trivially. Thus let us in the remainder of the proof assume that $i_m \notin MustBeAfter(i_0)$.

Now let us show that $i_0 \in CanBeBefore(i_j)$ must hold, which in turn (together with $i_0 \notin CanBeAfter(i_j)$) implies $i_0 \in MustBeBefore(i_j)$. Let us assume by contradiction that $i_0 \notin CanBeBefore(i_j)$. However, at the time when both $i_0 \notin CanBeAfter(i_j)$ and $i_0 \notin CanBeBefore(i_j)$ became true, that is, when the second of these conditions was made satisfied by rule /1/, rule /1/ must have posted the constraint $(Valid(i_0)=0 \vee Valid(i_j)=0)$ which contradicts the assumptions of the lemma. By a symmetric argument we can prove that $i_m \in MustBeAfter(i_j)$. Thus when rule /1/ is triggered by setting $Valid(i_j) \leftarrow 1$ both $i_0 \in MustBeBefore(i_j)$ and $i_m \in MustBeAfter(i_j)$ hold (and $i_m \notin MustBeAfter(i_0)$ is assumed), and therefore rule /1/ removes i_m from the set $CanBeBefore(i_0)$ as well as i_0 from the set $CanBeAfter(i_m)$, which finishes the proof.

Q.E.D.

Proposition 2: The worst-case time complexity of the propagation rule /1/ (instantiation of the Valid variable) including all possible recursive calls is $O(n^2)$, where n is the number of activities.

Proof: If activity A is made invalid then all exclusion pairs that include A are removed from set Ex which could be done in time $O(n)$, if the set is properly implemented (for example as a symmetric $n \times n$ matrix). Moreover, activity A it is removed from the sets CanBeBefore, CanBeAfter, CanBeRightBefore, and CanBeRightAfter of all other activities which takes the total time $O(n)$.

If activity A becomes valid then some activities are made invalid and some new arcs may be added to the graph. At most n activities can be invalidated which takes a total time $O(n^2)$. The maximal number of added arcs is $\Theta(n^2)$. It may also happen that some other activities (at most $O(n)$) become invalid to break cycles. However, we already know that the time complexity of making an activity invalid is $O(n)$. Together, the worst-case time complexity to make an activity valid is $O(n^2)$.

Q.E.D.

In some situations arcs may be added to the double precedence graph during the solving procedure, either by the user, by the scheduler/planner, or by other filtering algorithms [2]. The following rule /2/ updates the double precedence graph to keep transitive closure when an arc is added to the double precedence graph. If a new arc A«B is added then we first check whether the arc is not already present in the graph. If it is a

new arc then the corresponding sets are updated and a possible cycle is detected (we use the same reasoning as in rule /1/). Finally, if any end point of the arcs is valid, then necessary arcs are added to update the transitive closure according to Definition 1. In such a case, some direct precedence relations are removed according to Definition 2. Note that the propagators for new arcs are evoked after the propagator of the current rule finishes.

```
A«B is added →                                                          /2/
   if A∈MustBeBefore(B) then exit  // the arc is already present
   CanBeAfter(B) ← CanBeAfter(B) \ {A}
   CanBeBefore(A) ← CanBeBefore(A) \ {B}
   CanBeRightAfter(B) ← CanBeRightAfter(B) \ {A}
   CanBeRightBefore(A) ← CanBeRightBefore(A) \ {B}
   if A∉CanBeBefore(B) then    // break the cycle
    if Valid(A)=1 then Valid(B) ← 0  // Valid(B)=1 leads to fail
    else if Valid(B)=1 then Valid(A) ← 0  // Valid(A)=1 leads to fail
         else Ex ← Ex ∪ {{A,B}}
   else    // transitive closure
     if Valid(A)=1 then
       for each C∈MustBeBefore(A) s.t. Valid(C)≠0 do
         CanBeRightAfter(C) ← CanBeRightAfter(C) \ {B}
         CanBeRightBefore(B) ← CanBeRightBefore(B) \ {C}
         if C∉MustBeBefore(B) then
              add C«B
     if Valid(B)=1 then
       for each C∈MustBeAfter(B) s.t. Valid(C)≠0 do
         CanBeRightAfter(A) ← CanBeRightAfter(A) \ {C}
         CanBeRightBefore(C) ← CanBeRightBefore(C) \ {A}
         if C∉MustBeAfter(A) then
              add A«C
   exit
```

Again, it is possible to show that if the precedence graph G is transitively closed (in the sense specified by Definition 1) and arc A « B is added to G then rule /2/ updates the precedence graph G to be transitively closed again. Note also, that propagation rules /1/ and /2/ achieve global consistency concerning the precedence constraint. This is a direct consequence of keeping a transitive closure of the precedence graph.

Proposition 3: If the precedence graph G is transitively closed (in the sense specified by Definition 1) and arc A « B is added to G then rule /2/ updates the precedence graph G to be transitively closed again.

Proof: Assume that arc A « B is added into G at a moment when arc B « C is already present in G. Moreover assume that Valid(A)≠0, Valid(B)=1, and Valid(C)≠0. We want to show that A « C is in G after rule /2/ is fired by the addition of A « B. The presence of arc B « C implies that C∈MustBeAfter(B) (and by symmetry also B∈MustBeBefore(C)). Now there are two possibilities. Either C∉MustBeAfter(A) in which case rule /2/ adds the arc A « C into G, or C∈MustBeAfter(A) (and by symmetry also A∈MustBeBefore(C)) which means that arc A « C was already present in G when arc A « B was added.

The case when arc A « B is added into G at a moment when arc C « A is already present in G and Valid(C)≠0, Valid(A)=1, Valid(B)≠0 holds can be handled similarly.

Thus when an arc is added into G, all paths of length two with a valid midpoint which include this new arc are either already spanned by a transitive arc, or the transitive arc is added by rule /2/. In the latter case this may invoke adding more and more arcs. However, this process is obviously finite (cannot cycle) as an arc is added into G only if it is not present in G, and no arc is ever removed from G. More on the time complexity of arc additions follows in Proposition 4.

Therefore, it is easy to see, that when the process of recursive arc additions terminates, the graph G is transitively closed. Indeed, for every path of length two in G with a valid midpoint one of the arcs on the path is added later than the other, and we have already seen that at a moment of such an addition the transitive arc is either already in G or is added by rule /2/ in the next step.

Q.E.D.

Proposition 4: The worst-case time complexity of the propagation rule /2/ (adding a new arc) including all recursive calls to rules /1/ and /2/ is $O(n^3)$, where n is the number of activities.

Proof: If arc A«B is added and B must also be before A then one of the activities A or B may become immediately invalid which takes time $O(n)$ (see Proof of Proposition 2). If both A and B are undecided then the rule prunes sets CanBeAfter(B) and CanBeBefore(A) and exits without further propagation. If A is valid and B is undecided (or vice versa) then all predecessors of A are connected to B. There are at most $O(n)$ such predecessors and the new arcs are added by recursive invocation of rule /2/. The recursion stops at this level because every predecessor X of a valid predecessor C of A is also a predecessor of A (due to the transitive closure) and hence the arc X«B has already been enqueued for propagation when addition of A«B was processed. Moreover, any duplicate copy of the same arc in the queue will be processes in time O(1) (see the first line of rule /2/). The "worst" situation happens when both A and B are valid. Then all predecessors of A are recursively connected to all successors of B. There are at most $O(n^2)$ such connections and processing each connection takes time $O(n)$ – see the for loops in rule /2/, so the worst-case time complexity is $O(n^3)$.

Q.E.D.

Proposition 5: The rules /1/ and /2/ ensure that if B « A or there is a valid activity C between A and B (that is, A « C and C « B) then A∉CanBeRightBefore(B) and B∉CanBeRightAfter(A).

Proof: We will prove the proposition for the set CanBeRightBefore only, the set CanBeRightAfter is maintained symmetrically. At the beginning, the set CanBeRightBefore(B) contains all activities which is all right, because all activities are undecided. If A is deleted from CanBeBefore(B) (due to adding B « A), A is also deleted from CanBeRightBefore(B) in both rules /1/ and /2/. If any C becomes valid, A∈MustBeBefore(C), and B∈MustBeAfter(C) then A is deleted from CanBeRightBefore(B) in rule /1/. If a new arc A«C is added, C is valid, and B∈MustBeAfter(C) then A is deleted from CanBeRightBefore(B) in rule /2/. Similarly, if a new arc C«B is added, C is valid, and A∈MustBeBefore(C) then A is deleted from CanBeRightBefore(B) in rule /2/.

Q.E.D.

A Propagation Rule for Direct Precedences

So far we more or less ignored the restrictions imposed by the state transition diagram. The reason is that these restrictions can be easily encoded by removing explicitly direct precedence relations from the double precedence graph. In particular, if transition from A to B is forbidden by the state transition diagram then arc A «$_d$ B is removed from the double precedence graph. In a totally ordered set of activities it implies there must be some valid activity C between A and B or B must be after A. Actually a stronger requirement can be imposed: if A is before B (and A cannot be directly before B) then there must be some valid activity directly before B that is also after A and some valid activity directly after A that is before B. This observation can be transformed into the following implications:

$$\text{CanBeRightAfter}(A) \cap \text{CanBeBefore}(B) = \varnothing \Rightarrow B \ll A$$
$$\text{CanBeAfter}(A) \cap \text{CanBeRightBefore}(B) = \varnothing \Rightarrow B \ll A.$$

The above reasoning can be used to deduce a new precedence constraint B « A and, vice versa, if A « B then we can actively look for activities between A and B, especially, if there is only one candidate for such activity. This reasoning is realised using two propagation rules. First, the direct precedence is removed using rule /3/ and rule /4/ is activated. Rule /4/ is then called whenever there are some changes related to activities A or B. This rule tries to deduce that B must be before A or if A « B then the rule looks for some activity C between A and B.

```
A«dB is deleted →                                                    /3/
  CanBeRightAfter(A) ← CanBeRightAfter(A) \ {B}
  CanBeRightBefore(B) ← CanBeRightBefore(B) \ {A}
  activate rule /4/ for A and B
  exit

CanBeRightAfter(A) or CanBeAfter(A) or CanBeBefore(A) or
CanBeRightBefore(B) or CanBeBefore(B) or CanBeAfter(B) is changed,
or Valid(A) or Valid(B) is instantiated →                            /4/
  if Valid(A)=0 or Valid(B)=0 or A∈MustBeAfter(B) then exit
  if CanBeRightAfter(A)∩CanBeBefore(B)=∅
    or CanBeAfter(A)∩CanBeRightBefore(B)=∅ then
      add B«A
      exit
  if A∈MustBeBefore(B) & Valid(A)=1 & Valid(B)=1 then
     if {C}=CanBeRightAfter(A)∩CanBeBefore(B) then
       // C is the only possible direct successor of A
       add A«C
       add C«B
       Valid(C) = 1
       exit
     if {C}= CanBeAfter(A)∩CanBeRightBefore(B) then
       // C is the only possible direct predecessor of B
       add A«C
       add C«B
       Valid(C) = 1
       exit
```

If there are no explicit direct precedence relations like those imposed by the state transition diagram, then we already proved that propagation rules /1/-/2/ achieve global consistency. Unfortunately, global consistency cannot be achieved for rules /3/-/4/, that is, for explicitly removed direct precedence relations. Nevertheless, we can show that the constraint realised by rules /3/-/4/ is complete.

Proposition 6: If all activities are either valid or invalid and the set of valid activities is totally ordered then this order satisfies the restrictions imposed by the state transition diagram.

Proof: Assume for contradiction that there are valid activities A and B such that A is directly before B in the sequence but the state transition diagram forbids A to be directly before B. In such a case, rule /3/ has been called so $B \notin CanBeRightAfter(A)$ and rule /4/ is active. There is no invalid activity in CanBeRightAfter(A) due to rule /1/. For every valid activity C, either C « A or B « C and hence $C \notin CanBeRightAfter(A)$ due to rules /1/ or /2/. Recall that rule /4/ is called every time the set CanBeRightAfter(A) is changed. We just showed that $CanBeRightAfter(A)=\emptyset$ and therefore also $CanBeRightAfter(A) \cap CanBeBefore(B)=\emptyset$. Therefore the second condition in rule /4/ is true and hence B « A is deduced which leads to failure. The rule /4/ cannot exit using the first condition because A and B are valid and A « B. The rule also cannot exit using the third condition because then there is a valid activity C such that A « C and C « B which is in contradiction with the order if activities. In any case, rule /4/ deduces failure so A cannot be right before B in any solution.

Q.E.D.

Proposition 7: The worst-case time complexity of the propagation rule /4/ including all recursive calls to rules /1/ and /2/ is $O(n^3)$, where n is a number of activities.

Proof: The time complexity of propagation rule /4/ alone is $O(n)$ because the intersection operations may require this time. The rule can add at most four arcs and it can make two activities valid. According to Proposition 2, making activity valid requires time $O(n^2)$. According to Proposition 4 adding an arc (including all recursive calls) requires time $O(n^3)$. Hence the total worst-case time complexity is $O(n^3)$.

Q.E.D.

4 A Note on Open Graphs

The double precedence graphs studied in previous sections assume that the number of activities or at least its upper estimate is known. We use optional activities to deactivate activities that will not be part of the solution. This technique is appropriate in scheduling applications where most activities are known and optional activities are used to model alternatives to be decided during scheduling. However, in planning this technique is less convenient because the number of activities is unknown. It is still possible to use optional activities but in this case, the total number of activities will be probably too large which will decrease overall efficiency.

Our constraint model can be used directly to include new activities that will appear during problem solving. Recall, that we model the double precedence graph using difference sets, in particular the set CanBeBefore(A)\CanBeAfer(A) describes the activities that must be before A. We assumed that these sets are subsets of {1,…, n}, where n is the number of activities. To model problems where the number of activities is unknown in advance, we can use an infinite set {1,…,sup}, where sup is a computer representation of "plus infinity". The activities, that are already known, are represented using the variable Valid and sets CanBeBefore and CanBeAfter. The other activities are represented just by their indices in these sets. Hence, these activities behave like optional undecided activities with no precedence relations to activities already in the graph. Therefore, there is no propagation related to these activities so sets representing these activities are not changing and hence it is not necessary to keep them in memory (only indices of invalid activities may be deleted from these sets, but it does not play any role). As soon as a new activity is included in the precedence graph then an index is assigned to the activity and its set representation is created. At this time all invalid activities should be removed from the sets of the new activity. We only need to keep the number of activities already included in the precedence graph to know which index can be used. Note finally, that we can still use optional activities to model alternatives to be decided later.

In addition to adding activities from outside, it is possible to use the double precedence graph to deduce that a new activity must be added to the graph. In particular, if A « B but A cannot be directly before B and no existing activity is between A and B then we can deduce that a new activity C must be added together with the precedence relations A « C and C « B. This might be useful especially to resolve flaws in plan-space planning.

5 Experimental Results

We are currently working on implementation of the proposed filtering algorithms. We have already implemented the model of the precedence graph with optional activities in SICStus Prolog 3.12.3 using the standard interface for the definition of global constraints. In this section, we present some preliminary experimental results comparing our approach with the constraints model from [5] using min-cutset problems. The min-cutset problem consists of precedence relations and the task is to find the largest set of vertices such that the sub-graph induced by these vertices does not contain any cycle (or symmetrically to find the smallest set of vertices such that all cycles are broken if these vertices are removed from the graph). This problem is known to be NP-hard [7].

We use the data set from [12] to compare our approach based on the precedence graph with the CLP model from [5] based on absolute positioning in the sequence of activities (Original). All the problems in the data set consist of 50 activities while the number of precedence constraints varies. Table 1 shows the specification of problems used in our experiment and the best solutions obtained. Note that the solutions obtained by our approach (Precedence) are optimal. The experiments run under Windows XP Professional on 1.1 GHz Pentium-M processor with 1280 MB RAM.

Table 1. Min-cutset problems.

Bench	Activities	Precedences	Original best	Precedence best
P50-100	50	100	47	47
P50-150	50	150	41	41
P50-200	50	200	35	37
P50-250	50	250	31	33
P50-300	50	300	28	31
P50-500	50	500	21	22
P50-600	50	600	17	19
P50-700	50	700	16	17
P50-800	50	800	16	16
P50-900	50	900	14	14

Figure 3 shows the comparison of runtimes and the number of backtracks for both approaches. Our approach requires more than an order of magnitude less backtracks and less runtime to find the optimal solution. In fact, with the exception of problems with 50 and 100 precedence constraints, the original CLP model was not able to find the optimal solution (or to prove optimality) within the time limit of 50 minutes. Note finally, that concerning the runtime we cannot compete with the GRASP heuristic proposed in [12], but this was not our original ambition as we tackle different problems. Moreover, opposite to the GRASP approach our technique is complete and, indeed, for some problems we have found better solutions than reported in [12].

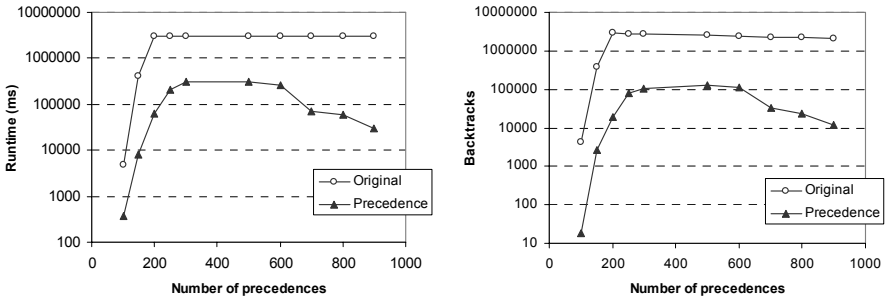

Fig. 3. Computation results on min-cutset problems

6 Conclusions

We introduced a new constraint model describing precedence graphs with optional activities and direct precedence relations. For this model we proposed propagation rules that keep a transitive closure of the graph and remove inconsistencies caused by forbidden direct precedence relations. If explicit direct precedences are not present then the proposed rules achieve global consistency. We also experimentally showed

that this model of the precedence graph is more efficient than a straightforward implementation of precedence relations. If explicit direct precedences, for example modelling state transition diagram, are present then the proposed rules realise a complete constraint model though the domain filtering is not complete. Rather than proposing a monolithic algorithm, we focused on incremental propagation of changes and on implementation-friendly architecture that is easy to translate into propagation rules usable in existing constraint solvers. Moreover this approach is extendable to problems where the number of activities in unknown in advance.

7 Acknowledgements

The research is supported by the Czech Science Foundation under the contract no. 201/04/1102.

References

1. Baptiste, P.; Le Pape, C.; and Nuijten, W.: *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer Academic Publisher, 2001.
2. Barták, R.: Incremental Propagation of Time Windows on Disjunctive Resources. In *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference* (FLAIRS 2006), pp. 25-30. 2006.
3. Beck J.Ch. and Fox M.S.: Scheduling Alternative Activities. *Proceedings of AAAI-99*, USA, pp. 680-687, 1999.
4. Cesta, A. and Stella, C.: A Time and Resource Problem for Planning Architectures, *Recent Advances in AI Planning*, LNAI 1348, Springer Verlag, 117-129, 1997.
5. Fages, F.: CLP versus LS on log-based reconciliation problems for nomadic applications. In *Proceedings of ERCIM/CompulogNet Workshop on Constraints*, Praha, 2001.
6. Focacci, F.; Laborie, P.; and Nuijten, W.: Solving Scheduling Problems with Setup Times and Alternative Resources. In *Proceedings of AIPS 2000*, 2000.
7. 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.
8. Laborie, P.: Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence* 143: 151–188, 2003.
9. Laborie, P.: Resource temporal networks: Definition and complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 948–953, 2003.
10. Moffitt, M. D.; Peintner, B.; and Pollack, M. E.: Augmenting Disjunctive Temporal Problems with Finite-Domain Constraints. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-2005)*. 1187–1192. AAAI Press, 2005.
11. Stergiou, K., and Koubarakis, M.: Backtracking algorithms for disjunctions of temporal constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, 248–253. AAAI Press, 1998.
12. Pardalos, P.M.; Qian, T.; Resende, M.G.: A greedy randomized adaptive search procedure for the feedback vertex set problem. *Journal of Combinatorial Optimization*, 2:399-412, 1999.