ICAPS05

WS1

Workshop on Constraint Programming for Planning and Scheduling

J. Christopher Beck
University of Toronto, CANADA

Andrew Davenport
IBM TJ Watson, USA

Toby Walsh
UNSW, Sydney and NICTA, AUSTRALIA

ICAPS 2005
Monterey, California, USA
June 6-10, 2005

CONFERENCE CO-CHAIRS:

Susanne Biundo
University of Ulm, GERMANY

Karen Myers
SRI International, USA

Kanna Rajan
NASA Ames Research Center, USA

Cover design: L.Castillo@decsai.ugr.es

Workshop on Constraint Programming for Planning and Scheduling

J. Christopher Beck
University of Toronto, CANADA

Andrew Davenport
IBM TJ Watson, USA

Toby Walsh
UNSW, Sydney and NICTA, AUSTRALIA










Workshop on Constraint Programming for Planning and Scheduling: CPPS05

Table of contents

Workshop on Constraint Programming for Planning and Scheduling: CPPS05

Preface

Constraint programming is a powerful technology for expressing and reasoning about real-world planning and scheduling problems. It permits a wide range of complex constraints, such as resource constraints, temporal constraints and sequencing constraints, to be compactly specified and efficiently reasoned about. The effective solution of planning and scheduling problems using constraint programming requires the use of ideas from a wide range of areas from AI and OR (including knowledge representation, temporal reasoning, reasoning under uncertainty, constraint relaxation and decomposition techniques). The aim of this workshop is to foster the introduction of more ideas from these and other areas.

Organizers

- *Chris Beck, University of Toronto, Canada*
- *Andrew Davenport, IBM TJ Watson, USA*
- *Toby Walsh , UNSW, Sydney and NICTA, Australia*

Programme Committee

- *Roman Bartak, Charles University, Czech Republic*
- *Amedeo Cesta, ISTC-CNR, Italy*
- *Vincent Cicirello, Drexel University, USA*
- *Bill Havens, Simon Fraser University, Canada*
- *Philippe Laborie, ILOG, SA, France*
- *Robert Morris, NASA, USA*
- *Francesca Rossi, University of Padova, Italy*
- *Stephen Smith, Carnegie Mellon University, USA*
- *Mark Wallace, Monash University, Australia*
- *Thierry Vidal, Ecole Nationale d'Ingenieurs de Tarbes, France*

Incremental Propagation Rules for Precedence Graph with Optional Activities and Time Windows

Roman Barták*, Ondřej Čepek*†

*Charles University
Faculty of Mathematics and Physics
Malostranské náměstí 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

Constraint-based scheduling is a powerful tool for solving real-life scheduling problems thanks to a natural integration of special solving algorithms encoded in global constraints. The filtering algorithms behind these constraints are based on propagation rules modelling some aspects of the problems, for example a unary resource. This paper describes new incremental propagation rules integrating a propagation of precedence relations and time windows for activities allocated to a unary resource. Moreover, the rules also cover so called optional activities that may or may not be present in the final schedule.

Introduction

Real-life scheduling problems usually include a variety of constraints so special scheduling algorithms (Brucker, 2001) describing a single aspect of the problem can hardly be applied to solve the problem completely. Constraint-based scheduling (Baptiste, Le Pape, Nuijten, 2001) provides a natural framework for modelling and solving real-life problems because it allows integration of different constraints. The above mentioned special scheduling algorithms can be often transformed into propagators for the constraints so the big effort put in developing these algorithms is capitalised in constraint-based scheduling.

Many filtering algorithms for specialised scheduling constraints have been developed in recent years (Baptiste, Le Pape, Nuijten, 2001). There exist algorithms based for example on edge-finding (Baptiste & Le Pape, 1996) or not-first/not-last (Torres & Lopez, 1997) techniques that restrict the time windows of the activities. Other algorithms are based on relative ordering of activities, for example filtering based on optimistic and pessimistic resource profiles (Cesta & Stella, 1997). Recently, as scheduling and planning technologies are coming together, filtering algorithms combining filtering based on relative ordering and time windows appeared. Detectable precedences by Vilím (2002) are one of the first attempts for such a combination. Laborie (2003) presents a similar rule called energy precedence constraint for reservoir-like resources.

Filtering algorithms for scheduling constraints typically assume that all the constrained activities will be included in the final schedule. This is not always true, for example assume that there are alternative processes to accomplish a job or alternative resources per activity. These alternatives are typically modelled using optional activities that may or may not be included in the final schedule depending on which process or resource is selected. The optional activity may still participate in the constraints but it should not influence other activities until it is known to be in the schedule. This could be realised by allowing the duration of the optional activity to be zero for time-windows based filtering like edge-finding (Baptiste, Le Pape, Nuijten, 2001). However, this makes filtering weaker and as shown in (Vilím, Barták, Čepek, 2004) a stronger and faster filtering can be achieved if optional activities are assumed in the filtering algorithm directly. The paper (Focacci, Laborie, Nuijten, 2000) proposed a global precedence graph where alternative resources correspond to paths in the graph, but the graph is used merely for cost-based filtering (optimization of makespan or setup times).

In this paper we address the problem of integrated filtering based on precedence relations and time windows. From the beginning we assume the existence of optional activities. A filtering algorithm for these so called detectable precedences with optional activities on a unary resource has been proposed in (Vilím, Barták, Čepek, 2004). This algorithm uses Θ-Λ-tree to achieve O(n.log n) time complexity and it is a monolithic algorithm (must be repeated completely if there is any change of domains). The same pruning can be achieved by the energy precedence constraint proposed by Laborie (2003) if it is applied to a unary resource (the energy precedence constraint is defined for reservoirs). However, the energy precedence constraint is not defined for optional activities and details of implementation are not given in the paper.

We propose a new set of propagation rules that keep a transitive closure of the precedence relations, deduce new precedence relations, and shrink the time windows of the activities. They may also deduce that some optional activity will not be present in the final schedule. There are two main differences from the algorithm proposed in (Vilím, Barták, Čepek, 2004). First, we use "light" data structures, namely domains of variables. Second, the new rules are incremental so they directly react to changes of particular domains rather than running a monolithic algorithm from scratch. Such rules are much easier for implementation and for integration to existing constraint solvers and the hope is their incremental nature will lead to a good practical efficiency. The implementation of the rules is currently being done so the paper reports a work in progress.

The paper is organised as follows. We first give more details on the problem to be solved. Then we describe the constraint services available for implementation of new constraints. In the main part of the paper, we describe a constraint-based representation of the precedence graph and we propose a set of propagation rules for the precedence graph. After that, we describe propagation rules for shrinking time windows by using information about precedence relations.

The Problem

In this paper we address the problem of modelling a unary resource where activities must be allocated in such a way that they do not overlap in time. We assume that there are time windows restricting the position of these activities. The time window [R,D] for an activity specifies that the activity cannot start before R (release time) and cannot finish after D (deadline). We assume the activity to be non-interruptible so the activity occupies the resource from its start till its completion, i.e. for a time interval whose length is equal to the given length of the activity. We also assume that that there are precedence constraints for the activities. The precedence constraint A«B specifies that activity A must not finish later than activity B starts. The precedence constraints describe a partial order between the activities. The goal of scheduling is to decide a total order that satisfies (extends) the partial order (this corresponds to the definition of a unary resource) in such a way that each activity is scheduled within its time window. Last but not least we allow some activities to be so called *optional*. It means that it is not known in advance whether such activities are allocated to the resource or not. If the optional activity is allocated to the resource, that is, it is included in the final resource schedule then we call this activity *valid*. If the activity is known not to be allocated to the resource then we call the activity *invalid*. In other cases, that is the activity is not decided to be or not to be allocated to the resource, we call the activity *undecided*. Optional activities are useful for modelling alternative resources for the activities (an optional activity is used for each alternative resource and exactly one optional activity becomes valid) or for modelling alternative processes to accomplish a job (each process may consist of a different set of activities).

Note that for the above defined problem of scheduling with time windows it is known that deciding about an existence of a feasible schedule is NP-hard in the strong sense (Garey & Johnson, 1979) even when no precedence relations or optional activities are considered, so there is a little hope even for a pseudo-polynomial solving algorithm. Hence using propagation rules and constraint satisfaction techniques is justified there.

Constraints and Constraint Services

Constraint satisfaction problem is defined as a triple (X,D,C), where X is a finite set of variables, D is a set of domains for these variables, each variable may have its own domain which is a finite set of values, and C is a set of constraints restricting possible combinations of the values assigned to variables (a constraint is a relation over the variables' domains). The task is to find a value for each variable from the corresponding domain in such a way that all the constraints are satisfied (Dechter, 2003).

There exist many constraint solvers that provide tools for solving constraint satisfaction problems, for example ILOG Solver, Mozart or the clpfd library of SICStus Prolog. These solvers are typically based on combination of domain filtering with depth-first search. Domain filtering is a process of removing values from the domains that do not satisfy some constraint. Each constraint has a filtering algorithm assigned to it that does this job for the constraint, and these algorithms communicate via the domains of the variables – if a filtering algorithm shrinks a domain of some variable, the algorithms for constraints that use this variable propagate the change to other variables until a fixed point is reached or until some domain becomes empty. Such a procedure is called a (generalised) arc consistency. When all domains are reduced to singletons then the solution is found. If some domain becomes empty then no solution exists. In all other cases the search procedure splits the space of possible assignments by adding a new constraint (for example by assigning a value to the variable) and the solution is being searched for in sub-spaces defined by the constraint and its negation (other branching schemes may also be applied).

The constraint solvers usually provide an interface for user-defined filtering algorithms so the users may extend the capabilities of the solvers by writing their own filtering algorithms (Schulte, 2002). This interface consists of two parts: triggers and propagators. The user should specify when the filtering algorithm is called – a *trigger*. This is typically a change of domain of some variable, for example when the lower bound of the domain is increased, the upper bound is decreased, or any element is deleted from the domain. The *propagator* then describes how this change is propagated to domains of other variables. The constraint solver provides procedures for access to domains of variables and for operations over the domains

(membership, union, intersection, etc.). The output of the propagator is a proposal how to change domains of other variables in the constraint. The algorithm may also deduce that the constraint cannot be satisfied (fail) or that the constraint is entailed (exit). We will describe the propagation rules in such a way that they can be easily transformed into a filtering algorithm in the above sense. Each propagation rule will consist of a trigger describing when the rule is activated and a propagator describing how the domains of other variables are changed.

Rules for the Precedence Graph

As we mentioned above, precedence relations are defined among the activities. These precedence relations define a precedence graph which is an acyclic directed graph where nodes correspond to activities and there is an arc from A to B if A«B. Frequently, the scheduling algorithms need to know whether A must be before B in the schedule, that is whether there is a path from A to B in the precedence graph. It is possible to look for the path each time such a query occurs. However, if such queries occur frequently then it is more efficient to provide the answer immediately, that is, in time O(1). This can be achieved by keeping a transitive closure of the precedence graph.

Definition 1: We say that a precedence graph G is *transitively closed* if for any path from A to B in G there is also an arc from A to B in G.

Defining the transitive closure is more complicated when optional activities are assumed. In particular, if A«B and B«C and B is undecided then we cannot deduce that A«C simply because if B is removed – becomes invalid – then the path from A to C is lost. Therefore, we need to define transitive closure more carefully.

Definition 2: 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 of the path length). Hence, if no optional activity is used (all activities are valid) then Definition 2 is identical to Definition 1.

In the next paragraphs we will propose a constraint model for the precedence graph and two propagation rules that maintain the transitive closure of the graph with optional activities. 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 not known yet 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 is a set of indices of activities that can be before a given activity. CanBeAfter is a set of indices of activities that can be after the activity. If we add an arc between A and B (A«B) then we remove the index of A from CanBeAfter(B) and the index of B from CanBeBefore(A). For simplicity reasons we will write A instead of the index of A. Note that these sets can be easily implemented as finite domains of two variables so a special data structure is not necessary. For this implementation we propose to include value 0 in above two sets to ensure that the domain is not empty even if the activity is first or last (an empty domain in CSP indicates the non-existence of a solution). The value 0 is not assumed as an index of any activity in the propagation rules. To simplify description of propagation rules we define the following sets (not kept in memory but computed on demand):

MustBeAfter = CanBeAfter \ CanBeBefore
MustBeBefore = CanBeBefore \ CanBeAfter
Unknown = CanBeBefore ∩ CanBeAfter.

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

We initiate the precedence graph in the following way. First, the variables Valid, CanBeBefore, and CanBeAfter with their domains are created. Then the known precedence relations are added in the above-described way (domains of CanBeBefore and CanBeAfter are pruned). Finally, the Valid variables for the valid activities are set to 1 (activities that are known to be invalid from the beginning may be omitted from the graph) and the following propagation rule is fired when Valid(A) is set.

The propagation rule is invoked when the validity status of the activity is known. "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.

```
Valid(A) is instantiated →                                   /1/
  if Valid(A) = 0 then
    for each B do /* disconnect A from B */
      CanBeBefore(B) ← CanBeBefore(B) \ {A}
      CanBeAfter(B) ← CanBeAfter(B) \ {A}
  else /* Valid(A)=1 */
    for each B∈MustBeBefore(A) do
      for each C∈MustBeAfter(A)\MustBeAfter(B) do
        /* new precedence B«C */
        CanBeAfter(C) ← CanBeAfter(C) \ {B}
        CanBeBefore(B) ← CanBeBefore(B) \ {C}
        if B∉CanBeBefore(C) then    // break the cycle
          post_constraint(Valid(B)=0 ∨ Valid(C)=0)
  exit
```

Observation: Note that rule /1/ maintains symmetry for all valid and undecided activities because the domains are pruned symmetrically in pairs. This symmetry can be

defined as follows: if Valid(B)≠0 and Valid(C)≠0 then B∈CanBeBefore(C) if and only if C∈CanBeAfter(B). This moreover implies that B∈MustBeBefore(C) if and only if C∈MustBeAfter(B).

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 (generalised) transitive closure according to Definition 2. To give a formal proof we need to define several notions more precisely.

Let J={0,1, … ,n} be the set of activities, where 0 is a dummy activity with the sole purpose to keep all sets CanBeAfter(i) and CanBeBefore(i) nonempty for all 1≤i≤n. Furthermore, let G=(J\{0},E) be the given precedence graph on the set of activities, and G^T=(J\{0},T) its (generalised) transitive closure (note that the previously used notation i«j does not distinguish between the arcs which are given as input and those deduced by transitivity). The formal definition of the set T can be now given as follows:

1. if (i,j)∈E then (i,j)∈T
2. if (i,j)∈T and (j,k)∈T and Valid(i)≠0 and Valid(j)=1 and Valid(k)≠0 then (i,k)∈T

Furthermore, the set T is not maintained as a list of pairs of activities. Instead, it is represented using the set variables CanBeAfter(i) and CanBeBefore(i), 1≤i≤n in the following manner: (i,j)∈T if and only if i∉CanBeAfter(j) and j∉CanBeBefore(i). The incremental construction of the set T can be described as follows.

Initialization: for every i ∈ J\{0} set
- CanBeAfter(i) ← J\{i}
- CanBeBefore(i) ← J\{i}
- Valid(i) ← {0,1}

Set-up: for every arc (i,j)∈E set
- CanBeAfter(j) ← CanBeAfter(j)\{i}
- CanBeBefore(i) ← CanBeBefore(i)\{j}

Propagation: whenever a variable is made valid, call rule /1/

Clearly, T is empty after the initialization and T=E after the set-up. Now we are ready to state and prove formally that rule /1/ is sufficient for maintaining the set T on those activities which are already valid or still undecided.

Proposition 1: Let $i_0, i_1, \dots, i_m$ be a path in E such that Valid(i_j)=1 for all 1≤j≤m-1 and Valid(i_0)≠0 and Valid(i_m)≠0 (that is, the endpoints of the path are both either valid or undecided and all inner points of the path are valid). Then (i_0,i_m)∈T, that is i_0∉CanBeAfter(i_m) and i_m∉CanBeBefore(i_0).

Proof: We shall proceed by induction on m. The base case m=1 is trivially true after the set-up. 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≤j≤m-1 be an index such that Valid(i_j)←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∉CanBeAfter(i_j) and i_j∉CanBeBefore(i_0) using the path $i_0, \dots, i_j$
- i_j∉CanBeAfter(i_m) and i_m∉CanBeBefore(i_j) using the path $i_j, \dots, i_m$

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

Now let us show that i_0∈CanBeBefore(i_j) must hold, which in turn (together with i_0∉CanBeAfter(i_j)) implies i_0∈MustBeBefore(i_j). Let us assume by contradiction that i_0∉CanBeBefore(i_j). However, at the time when both i_0∉CanBeAfter(i_j) and i_0∉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 ∨ Valid(i_j)=0) which contradicts the assumptions of the lemma. By a symmetric argument we can prove that i_m∈MustBeAfter(i_j). Thus when rule /1/ is triggered by setting Valid(i_j)←1 both i_0∈MustBeBefore(i_j) and i_m∈MustBeAfter(i_j) hold (and i_m∉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.

From now on there will be no need to distinguish between the "original" arcs from E and the transitively deduced ones, so we will work solely with the set T. To simplify notation we shall switch back to the A«B notation (which is equivalent to (A,B) ∈ T).

In some situations arcs may be added to the precedence graph during the solving procedure, either by the user, by the scheduler, or by other filtering algorithms like the one described in the next section. The following rule updates the precedence graph to keep transitive closure when an arc is added to the precedence graph.

```
A«B is added →                                                    /2/
  CanBeAfter(B) ← CanBeAfter(B) \ {A}
  CanBeBefore(A) ← CanBeBefore(A) \ {B}
  if A∉CanBeBefore(B) then    // break the cycle
    post_constraint(Valid(A)=0 ∨ Valid(B)=0)
  else
    if Valid(A)=1 then    // transitive closure
      for each C∈MustBeBefore(A)\MustBeBefore(B) do
        add C«B
    if Valid(B)=1 then    // transitive closure
      for each C∈MustBeAfter(B)\MustBeAfter(A) do
        add A«C
  exit
```

The rule /2/ does the following. If a new arc is added then the sets CanBeBefore and CanBeAfter are updated. If a cycle is detected then the cycle is broken in the same way

as in rule /1/. The rest of the propagation rule ensures that if an arc is added and one of its endpoints is valid then other arcs are added recursively to keep a transitive closure. The following proposition shows that all necessary arcs are added by rule /2/.

Proposition 2: If the precedence graph is transitively closed and some arc is added then the propagation rule /2/ updates the precedence graph to be transitively closed again.

Proof: If an arc A«B is added and B is valid then according to the definition of transitive closure for each C such that B«C the arc A«C should be present in the precedence graph. The rule /2/ adds all these arcs. Symmetrically, if A is valid then for each C such that C«A all arcs C«B (where A«B) are added by the rule. Note also, that if the rule adds a new arc then this change in the precedence graph is propagated further so it may force adding other arcs. Hence all the necessary arcs are added. The rule adds only new arcs so the recursive calls to the rule must stop sometime.

Rules for Time Windows

An absolute position of the activity in time is frequently restricted by a *release time* and *deadline* that define a *time window* for processing the activity. The activity cannot start before the release time and it must be finished before the deadline. We assume the activity to be uninterruptible so it occupies the resource from its start till its completion. The processing time of activity A is constant, we denote it by p(A). The goal of time window filtering is to remove time points from the time window when the activity cannot be processed. Usually, only the lower and upper bounds of the time window change so we are speaking about shrinking the time window.

The standard constraint model for time allocation of the activity assumes two variables – start(A) and end(A) – describing when the activity A starts and completes. Initially, the domain for the variable start(A) is [release_time(A), deadline(A)-p(A)] and, similarly, the initial domain for the variable end(A) is [release_time(A)+p(A), deadline(A)]. If these two initial domains are empty then the activity is made invalid. We will use the following notation to describe bounds of the above domains:

est(A) = min(start(A))	earliest start time
lst(A) = max(start(A))	latest start time
ect(A) = min(end(A))	earliest completion time
lct(A) = max(end(A))	latest completion time

This notation can be extended in a natural way to sets of activities. Let Ω be a set of activities, then:

$est(\Omega) = \min\{est(A), A\in\Omega\}$
$lst(\Omega) = \max\{lst(A), A\in\Omega\}$
$ect(\Omega) = \min\{ect(A), A\in\Omega\}$
$lct(\Omega) = \max\{lct(A), A\in\Omega\}$
$p(\Omega) = \sum\{p(A), A\in\Omega\}$

During propagation, we will be increasing est and decreasing lct which corresponds to shrinking the time window for the activity. For simplicity reasons we use a formula est(A) ← X to describe a requested change of est(A) which actually means est(A) ← max(est(A), X). Similarly lct(A) ← X means lct(A) ← min(lct(A), X).

The time windows can be used to deduce a new precedence between activities. In particular, if est(A)+p(A)+p(B)>lct(B) then activity A cannot be processed before activity B and hence we can deduce B«A. This is called a *detectable precedence* in (Vilím, 2002). Vice versa, the precedence graph can be used to shrink time windows of the activities. In particular, we can compute the earliest completion time of the set of valid activities that must be processed before some activity A and the latest start time of the set of valid activities that must be processed after A. These two numbers define bounds of the time window for A. Formally:

$est(A) \leftarrow \max\{est(\Omega)+p(\Omega) \mid \Omega\subseteq\{X|X«A \ \&\ Valid(X)=1\}\}$
$lct(A) \leftarrow \min\{lct(\Omega)-p(\Omega) \mid \Omega\subseteq\{X|A«X \ \&\ Valid(X)=1\}\}$

The above two formulas are special cases of the *energy precedence constraint* (Laborie, 2003) for unary resources. Note also that the new bound for est(A) can be computed in $O(n.\log n)$ time, where n is the number of activities in $\Theta = \{X \mid X«A \ \&\ Valid(X)=1\}$, rather that exploring all subsets $\Omega\subseteq\Theta$. The algorithm is based on the following observation: if Ω' is the set with the maximal est(Ω')+p(Ω') then $\Omega'\supseteq\{X \mid X\in\Theta \ \&\ est(\Omega')\le est(X)\}$, otherwise adding such X to Ω' will increase est(Ω')+p(Ω'). Consequently, it is enough to explore sets $\Omega_X = \{Y \mid Y\in\Theta \ \&\ est(X)\le est(Y)\}$ for each $X\in\Theta$ which is done by the following algorithm (the new bound is computed in the variable *end*):

dur ← 0
end ← inf
for each $Y\in\{X \mid X«A \ \&\ Valid(X)=1\}$
    in non-increasing order of est(Y) **do**
  dur ← *dur* + p(Y)
  end ← max(*end*, est(Y)+*dur*)

The bound for lct(A) can be computed in a symmetrical way in $O(n.\log n)$ time, where n is the number of activities in $\{X \mid A«X \ \&\ Valid(X)=1\}$.

We now present two groups of propagation rules working with time windows and a precedence graph. The first group of rules realise the energy precedence constraint in an incremental way by reacting to changes in the precedence graph. The rules are invoked by making the activity valid /1a/ and by adding a new precedence relation /2a/. Because these rules have the same triggers as the rules for the precedence graph, they can be actually combined with them. Hence, we name the new rules using the number of the corresponding rule for the precedence graph.

The rules shrink the time windows using information about the precedence relations as described above. Only valid activities influence time windows of other (non invalid) activities. This corresponds to our requirement that optional activities that are not yet known to be valid should not influence other activities but they can be influenced. Notice also that if A«C, C«B, and C is valid then it is enough to explore possible increase of est(C) only. The reason is that if est(C) is really increased then the rule /3/ is invoked for C (see below) and the change is propagated directly to est(B). Similarly, only activities B such that there is no valid activity C in between B and A are explored for change of lct(B).

When activity A becomes valid and B is after A (A«B) or when A is valid and arc A«B is added then A can (newly) participate in sets Ω_X that are used to compute est of B (see above). Visibly, only sets containing A are of interest because only these sets can lead to change of est(B). The other sets Ω_X used to update est(B) have already been explored or will be explored when calling the rules for some valid activity in Ω_X. Moreover, all valid activities C such that C«A are used to compute est(A) so they can complete together no later than in est(A). Hence these activities do not influence directly est(B) (they influence it through changes of est(A)). Thus, we need to explore all subsets of valid activities X such that X«B and ¬X«A and these subsets contain A. Only these subsets can deduce a possible change of est(B). These are exactly the sets used in rules /1a/ and /2a/. A symmetrical analysis can be done for activities B before A. Note also that sets Ω' in rules /1a/ and /2a/ can be explored in the same way as we described for the energy precedence constraint above.

```
Valid(A) is instantiated →                                   /1a/
  if Valid(A)=1 then
    for each B∈MustBeAfter(A) s.t.
          ¬∃C Valid(C)=1 & A«C & C«B do
      let Ω = {X | X∈MustBeBefore(B) &
                X∈CanBeAfter(A) & Valid(X)=1}
      est(B) ← max{est(Ω'∪{A})+p(Ω')+p(A) | Ω'⊆Ω}
    for each B∈MustBeBefore(A) s.t.
          ¬∃C Valid(C)=1 & B«C & C«A do
      let Ω = {X | X∈MustBeAfter(B) &
                X∈CanBeBefore(A) & Valid(X)=1}
      lct(B) ← min{lct(Ω'∪{A})-p(Ω')-p(A) | Ω'⊆Ω}
  exit

A«B is added →                                               /2a/
    if Valid(A)=1 & Valid(B)≠0 then
      let Ω = {X | X∈MustBeBefore(B) &
                X∈CanBeAfter(A) & Valid(X)=1}
      est(B) ← max{est(Ω'∪{A})+p(Ω')+p(A) | Ω'⊆Ω}
    if Valid(B)=1 & Valid(A)≠0 then
      let Ω = {X | X∈MustBeAfter(A) &
                X∈CanBeBefore(B) & Valid(X)=1}
      lct(A) ← min{lct(Ω'∪{B})-p(Ω')-p(B) | Ω'⊆Ω}
    exit
```

The second group of rules is triggered by shrinking the time window (/3/ for increased est and /4/ for decreased lct). The rules in this group can deduce that the activity is invalid, if it has an empty time window, and they can deduce a new detectable precedence. Moreover, if the activity is valid then the change of its time window is propagated to other activities whose relative position to a given activity is known (they are before or after the given activity). If est of valid activity A is increased then it may influence est of B such that A«B (note that B is either valid or undecided, because invalid activities are disconnected from the graph). This happens if and only if $est(B) \leq est(\Omega_A)+p(\Omega_A)$ (see above for the definition of Ω_A with respect to B). Notice that rule /3/ computes $est(\Omega_A)+p(\Omega_A)$ to update est(B). Symmetrically, rule /4/ updates lct(B) for activities B such that B«A, if necessary. Hence, the propagation rules incrementally maintain the energy precedence constraint.

```
est(A) is increased →                                        /3/
  if Valid(A)=0 or est(A)+p(A) > lct(A) then
    Valid(A) ← 0
    exit
  else
    ect(A) ← est(A)+p(A)
    for each B∈Unknown(A) do
      if est(A)+p(A)+p(B) > lct(B) then
        B«A      /* detectable precedence */
    if Valid(A)=1 then
      for each B∈MustBeAfter(A) s.t.
                ¬∃C Valid(C)=1 & A«C & C«B do
        est(B) ← est(A)+p(A)+
                ∑{p(X) | X∈MustBeBefore(B) &
                        est(A)≤est(X) & Valid(X)=1}

lct(A) is decreased →                                        /4/
  if Valid(A)=0 or est(A)+p(A) > lct(A) then
    Valid(A) ← 0
    exit
  else
    lst(A) ← lct(A)-p(A)
    for each B∈Unknown(A) do
      if est(B)+p(B)+p(A) > lct(A) then
        A«B      /* detectable precedence */
    if Valid(A)=1 then
      for each B∈MustBeBefore(A) s.t.
                ¬∃C Valid(C)=1 & B«C & C«A do
        lct(B) ← lct(A)-p(A)-
                ∑{p(X) | X∈MustBeAfter(B) &
                        lct(X)≤lct(A) & Valid(X)=1}
```

Conclusions

The paper reports a work in progress on constraint models for the unary resource with precedence relations between the activities and time windows for the activities. Optional activities that may or may not be allocated to the resource are also assumed. We propose a set of propagation rules

that keep a transitive closure of the precedence relations, deduce additional precedence constraints based on time windows, and shrink the time windows for the activities. These rules are intended to complement the existing filtering algorithms based on edge-finding etc. to further improve domain pruning. Our next steps include formal complexity analysis, detail comparison to existing propagation rules (edge finder, etc.), implementation of the proposed rules, and testing in real-life environment.

Acknowledgements

The research is supported by the Czech Science Foundation under the contract no. 201/04/1102. We would like to thank anonymous reviewers for comments on early draft.

References

Baptiste, P. and Le Pape, C. 1996. Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling, *Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group (PLANSIG)*.

Baptiste P., Le Pape C., and Nuijten W. 2001. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problem*s, Kluwer Academic Publishers.

Brucker P. 2001. *Scheduling Algorithms*, Springer Verlag.

Cesta A. and Stella C. 1997. A Time and Resource Problem for Planning Architectures, *Recent Advances in AI Planning (ECP'97)*, LNAI 1348, Springer Verlag, 117-129.

Dechter R. 2003. *Constraint Processing*, Morgan Kaufmann.

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

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

Laborie P. 2003. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143, 151-188.

Schulte C. 2002. Programming Constraint Services, High-Level Programming of Standard and New Constraint Services, Springer Verlag.

Torres P. and Lopez P. 1999. On Not-First/Not-Last conditions in disjunctive scheduling, *European Journal of Operational Research*, 127, 332-343.

Vilím P. 2002. Batch Processing with Sequence Dependent Setup Times: New Results, *Proceedings of the 4th Workshop of Constraint Programming for Decision and Control, CPDC'02*, Gliwice, Poland.

Vilím P., Barták R., and Čepek O. 2004. Unary Resource Constraint with Optional Activities, *Principles and Practice of Constraint Programming (CP 2004)*, LNCS 3258, Springer Verlag, 62-76.

Extending a Scheduler with Causal Reasoning: a CSP Approach

Simone Fratini, Amedeo Cesta and Angelo Oddi

Planning and Scheduling Team [PST]
ISTC-CNR — Institute for Cognitive Science and Technology — Italian National Research Council
Viale Marx 15, I-00137 Rome, Italy — name.surname@istc.cnr.it

Abstract

A scheduling problem consists in a set of pre-defined activities that have to be temporally situated with respect to a set of resource availability constraints. Constraint-based approaches to scheduling have achieved mature development. We are currently studying how a constraint based scheduler can be endowed with the ability to synthesize new activities, i.e., by reasoning on planning knowledge. This paper describes some steps in the direction of uniformly managing planning as a special case of task scheduling. The general aim is to solve integrated planning and scheduling problems by rationally integrating in the same architecture various of the state-of-the-art CSP algorithms.

Introduction

A planning problem specifies a *domain theory* concerning how changes may occur in the world. A plan reasoner manipulates *cause-effect* relations in the domain theory which is often encoded in the form of operators, e.g., in PDDL (Ghallab *et al.* 1998; Fox & Long 2003). This operators represent the actions performed by the executing agent in the environment in order to obtain the desired change. Independently of the "shape" of this knowledge it is important to remember that planning knowledge represents the causal theory that describes the "correct domain evolution". Reasoning in planning is mostly dedicated to such logical knowledge (which we here call "causal constraints"). In a scheduling problem a set of pre-defined activities have to be temporally situated with respect to a set of resource availability constraints. In representing and solving such problems the *temporal* and *resource* constraints play a key role.

In certain application domains the subdivision of the two problems as separate entities is quite motivated (see for example (Srivastava, Kambhampati, & Do 2001)). In other domains such a clear separation of the planning and scheduling phase is more questionable and architectural approaches to integrate the two problems have been developed. For instance O-PLAN (Currie & Tate 1991), IxTeT (Laborie & Ghallab 1995), HSTS (Muscettola *et al.* 1992), RAX-PS (Jonsson *et al.* 2000), or ASPEN (Chien *et al.* 2000)) have already succeeded in including aspects from both Planning and Scheduling (P&S) among their features. These architectures have always emphasized the use of a rich representation planning language to capture complex characteristics of the domain involving time and resource constraints.

Most of recent research on planning has been devoted to the integrated problem by evolving the plan domain specification formalism to include temporal duration specification and, with some restrictions, resource consumption (PDDL2.1 (Fox & Long 2003) is the standardized language to compare all the efforts in this direction). The pursued idea consists of stretching the specification of a pure causal/logical problem to include time and resource features.

In the present work we are trying to follow a rather opposite perspective: we start from a pure scheduling specification and introduce language primitives to specify causal constraints. The aim is to be able to specify a problem in which not all the activities are specified and some of them can be synthesized according to the particular choices done either to serve resource constraints or to represent particularly rich domains. This point of view of extending scheduling engines with some activity synthesizing capabilities has attracted a high attention especially to manage complex process environments (see for instance Visopt ShopFloor system (Bartak 2003)).

This very broad idea is currently implemented in a prototypical solver called OMP. By means of a constraint based representation, OMP uniformly deals with causal and resource constraints "on top" of a shared layer representing temporal information as a Simple Temporal Problem (STP) (Dechter, Meiri, & Pearl 1991). For the causal reasoning we use a representation of domain components (called *state variables*), consisting in temporal automata, as first proposed in HSTS (Muscettola *et al.* 1992; Muscettola 1994) and studied also in subsequent works (Cesta & Oddi 1996; Jonsson *et al.* 2000; Frank & Jonsson 2003).

In this architecture activities that have to be scheduled are organized as a network, where nodes represents activities and edges represents quantitative temporal constraints between them. Activities no longer represent a "blind set" of entities that someone produced and gave to the scheduler, but they maintain information about logical links between them. Thus, an integrated P&S architecture, can manage at the same time both types of information (causal and resource) and solve the whole problem of generating and scheduling these networks.

This approach can be usefully applied to those domains where the scheduling problem is actually the hardest aspect, resource reasoning is very critical with respect to causal

reasoning and requires specialized scheduling technologies. Those domains cannot be afforded with a planner that integrates some generic scheduling features. Indeed this kind of domain often does not require strong planning capabilities. But enhancing the solver with some planning capabilities allows us to tackle problems in these domains in a more flexible way, as we will show also with a practical example.

In this paper we describe how using CSP technology we have been able to (1) put causal reasoning into a scheduling framework, (2) model and (3) solve the corresponding planning and scheduling problems. We additionally present a sketchy description of OMP and describe its current solving abilities.

Scheduling with Causal Reasoning

As we said in the previous section, we focus on *causal reasoning* as a distinguishing factor between planning and scheduling. We start from a scheduling framework which is able to manage temporal and resource constraints, and try to understand how to increase the capabilities of this framework with planning features.

The approach described here relies on a constraint-based representation for scheduling problems and aims at describing a framework where both planning and scheduling problem instances have as a common representation model, the Constraint Satisfaction Problem (CSP) (Tsang 1993). A CSP consists in a set of variables $X = \{X_1, X_2, \ldots, X_n\}$ each associated with a domain D_i of values, and a set of constraints $C = \{C_1, C_2, \ldots, C_m\}$ which denote the legal combinations of values for the variables s.t. $C_i \subseteq D_1 \times D_2 \times \cdots \times D_n$. A solution consists in assigning to each variable one of its possible values so that all the constraints are satisfied. The resolution process can be seen as an iterative search procedure where the current (partial) solution is extended on each cycle by assigning a value to a new variable. As new decisions are made during this search, a set of *propagation rules* removes elements from the domains D_i which cannot be contained in any feasible extension of the current partial solution.

In a typical scheduling problem, there is a set of *activities* that require certain amounts of *resources*. There are also a set of temporal constraints between these activities (duration and minimal and maximal separation between pairs of activities). Thus the problem is to find *when* to start each activity in order to ensure that all temporal constraints are respected and resources are never over or under used, a constraint due to their finite capacity. Such a scheduling problem can be solved for example with a *Precedence Constraints Posting* (PCP) approach (Cheng & Smith 1994; Cesta, Oddi, & Smith 2002), in fact building a temporal network where start and end points for each activity are mapped as time points. The underlying CSP model for temporal information is usually an STP. Reasoning about *resource profiles* it is possible to deduce a set of additional *precedence constraints* between activities that, when posted, ensure resources are never over used. The problem solving approach is sketched in fig.1.

As we said the scheduling problem can be seen as the output of a planning step, when two steps, planning and scheduling, are serialized, and the activity network come from causal reasoning while temporal separation constraints between activities come from cause-effect relations between planned actions. Our aim is to go a little bit further than a serialization of these two steps, and present a way to model a scheduling domain where the modeler specifies how activities that have to be scheduled are linked to each other via cause-effect relationships, then find a solution for this problem, i.e. planning which activities have to appear in the scheduling problem, according to some *goals* that have to be achieved.

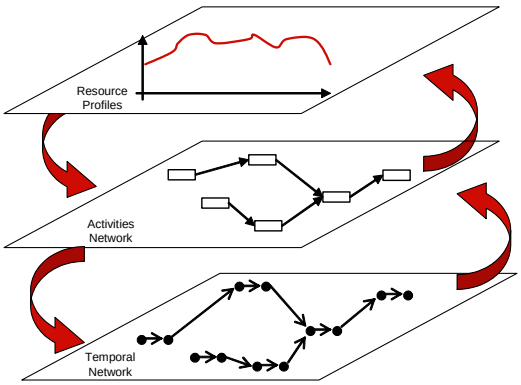


Figure 1: Activities Network Scheduling

Most of current planning frameworks employ the STRIPS ontology for domain description (Fikes & Nilsson 1971). The basic aspect of this type of problem formalization consists in describing domain causality in the form of actions performed by an executor. Actions explicitly describe the changes that their execution causes on the external world.

On the contrary, we follow a different ontological paradigm. Rather than focusing on the executing agent, we consider the relevant sub-parts of a domain that continuously evolves over time. Then instead of specify which action can be performed to modify the state of the world, under which conditions and with which effects, we directly specify which sequences of states are logically admissible for these sub-parts. The state of the entire world at each instant of time is the union of the values of these sub-parts at that instant. We call these sub-parts *state variables* because in fact there are entities, whose values over a temporal interval, determine what is going on on the world, in the form of temporally ordered sequences of state transitions. In addition to state variables, we use the notion of *resources* to model typical scheduling features, besides cause-effect relationships.

We consider here an example from a space environment where a spacecraft has to achieve some goals with its payloads, like taking pictures with a camera or gathering some specific data with instruments. A simplified model for this domain, where observations have to be stored into an internal memory then downloaded to Earth, can be represented in our framework using two resources, *Memory* and *Downlink_Channel*, a state variable *Satellite*, and a set of state variables $\{I_1, \ldots, I_n\}$ one for each on board instrument. The state variable *Satellite* should take values in

the set $\{Locked(target), Locking(target), Unlocked()\}$ where $Locked(target)$ is its value when the satellite is pointed toward the object $target$, $Locking(target)$ when the satellite is moving to point its target and $Unlocked()$ is a sort of idle status. Each instrument I_i takes values $Perform\_Observation(target, data)$ when an observation of object $target$ is performed and information is stored in $data$, $Download\_Memory(data)$ when $data$ are downloaded to Earth (using communication channels) and $Idle()$, an idle status.

As we have seen, a scheduling problem can be modeled over a temporal network, as a network of activities with a start and an end time point, linked to each other with temporal constraints, where each activity requires a certain amount of some resource. In a similar way, calling *task* a temporal interval with a start and an end time point, a planning problem can be seen as a network of tasks, linked to each other with temporal constraints, where each task says that the state variable must take between its start and end point one of the values specified. For instance in the bottom part of figure 2 we show an example of a task network for the state variable *Satellite* we introduced before. There are 3 tasks, two of them with only one allowable value. This network models a situation in which the state variable must take the value *Locking(x)* for 3 time units[1] in the interval $[t_1, t_3]$, the value *Locked(x)* for 2 time units in $[t_4, t_6]$ and one value in the set $\{Locking(x), Locked(x)\}$ for 3 time units in $[t_2, t_5]$. As in scheduling we reason on resource usages by summing activity requirement at each time point and calculating resource profiles. Likewise we can calculate a sort of "state variable profile" by intersecting for each time instant the values required by each task that overlaps in that time instant. In the top of figure 2 we show this profile. In fact the state variable *Satellite*, according with the task network, can take the value *Locking(x)* in $[t_1, t_3]$, both values *Locking(x)* or *Locked(x)* in $[t_3, t_4]$, the value *Locked(x)* in $[t_4, t_6]$ while it can be any value in its domain everywhere else.

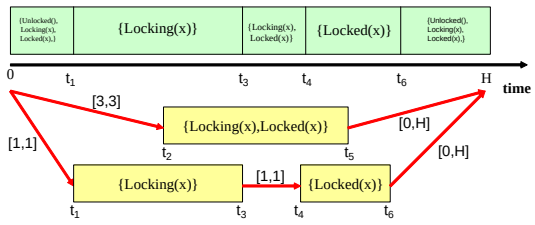


Figure 2: Tasks Network

With this representation of the problem we can see causal reasoning as a sort of *task scheduling*. The problem reduces to searching for a feasible ordering for these tasks taking care of the fact that each state variable needs to take at least one value at each instant. Thus a state variable is a sort of resource, where two tasks cannot overlap, unless their intersection is a non empty set of values.

[1]For the sake of simplicity duration constraints are not showed in the figure. We showed only temporal constraints that must occur between these tasks.

Moreover, because *tasks* and activities in fact share the same temporal network, the integration of P&S is achieved by mixing temporal constraints in an environment where two specialized reasoners, a scheduler for resources and a *task scheduler* for state variables, analyze the situation of a temporal constraint database, in fact posting constraints that affect the whole integrated problem. Hence our point of view: state variables and resources can be seen as *concurrent threads* in execution on a concurrent system, where a shared temporal model allows crossing relations between causal and resource usage aspects, even if two distinct reasoners affect them. In figure 3 we show three different networks, that can be either activities or tasks networks. They are connected with cross temporal constraints (dashed lines in the figure) which link (1) the causal problem with the scheduling problem (for instance requiring a resource usage when a task is performed by a state variable); (2) causal problems on different state variables (for instance requiring that when a task is performed by a state variable another state variable must perform some other tasks). Sharing a common temporal network they share the same temporal model, thus resources and state variable profiles in fact lie in the same temporal line, as concurrent threads. But at the same time causal, resource and temporal reasoning are clearly separated: the last one is represented in bottom layer in the figure, while the planning and scheduling problems are represented as networks in the middle layer. Finally we schedule these networks reasoning on resource and state variable profiles (represented in the top layer).

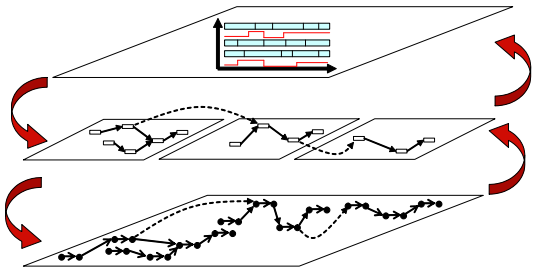


Figure 3: Tasks and Activities Integration

Coming back to our example, we want to model both a planning and a scheduling problem. A planning problem because performing observations requires the execution of some non trivial operations, such as pointing the satellite toward the object that has to be observed, *then* storing information *while* the satellite is locked, *then* downloading this information *when* Earth is visible from the satellite. And also a quite hard scheduling problem, because usually satellites have limited resources, little memory and not so wide communication channels. Thus we must pay attention to communication channel usage and memory management.

In order to model these causal constraints we must require *when* a state variable I_i takes the value $Perform\_Observation(target, data)$, the state variable $Satellite$ is $Locked(target)$ and a value $Download(data)$ occurs sometimes *after* the observation is performed. Moreover, because we want to download data to Earth,

when the value of a state variable I_i is $Download(Data)$, the state variable $Satellite$ must be $Locked(\text{“}Earth'')$ (that can be for instance a dashed crossing line in figure 3). Finally the satellite must move toward its target before pointing it, thus each state $Locked(target)$ has to be immediately preceded by a $Locking(target)$ state. This is a sort of domain theory, in which we specify how the application domain works by expressing a set of constraints over values that these variables can take over time. Activities that have to be scheduled over resources $Memory$ and $Downlink\_Channel$ are produced when an observation is performed (consuming memory) and when data are downloaded (freeing memory and using communication channels). Then we must require that when a state variable I_i takes a value $Perform\_Observation$ a resource consumption for resource $Memory$ has to be scheduled (another crossing line in figure 3 for instance), while a production for the same resource and a consumption/production pair for the resource $Downlink\_Channel$ must be scheduled. Of course the entity of such resource usages depends on how many data have to be stored or downloaded. We will explain more in detail in the next section this model.

It is worth pointing out that, due to its ontology, because a state variable can take no more than a single value for each time instant, we are also modeling the fact that the satellite can be pointed only toward an object at a time, and if we want to observe more than one object or download data, we have to change its pointing status, in fact moving the physical object that the state variable $Satellite$ models.

From this point of view, once we specify some goals we want to achieve, as for instance in our example defining some *tasks* as $Perform\_Observation(\text{“}Mars'')$ with an instrument I_1 between two time points t_1 and t_2, in order to follow all constraints specified in the domain theory, several other tasks that have to be allocated over state variables and several activities that have to be allocated over resources are automatically produced, generating task and activity networks such as those we saw before, that share the same temporal information. Thus, once all networkss are generated, in fact unfolding the domain theory and ensuring all cause-effect relationships are guaranteed according to the domain theory, the problem becomes to find a feasible ordering for these tasks and activities in a way that each state variable takes at least and no more than one value at each instant of time and resources are never over used. This purpose can be achieved by scheduling tasks and activities. Moreover propagation rules can be applied, in fact solving the whole problem as a CSP, alternating propagation and decision steps, as we will show more in detail later.

Knowledge Engineering

As we have seen before, we follow an ontological paradigm based on splitting the domain in several components, that can be state variables or resources, then specifying a domain theory as a set of constraints. Then, starting from some goals we want to achieve, producing networks of tasks and activities that have to be scheduled over state variables and resources.

Expressing a domain theory as a set of constraints over values that several components can assume suggested us to specify these components as a sort of *timed automata*. Each component can be specified as an automaton, where labeled state transition rules specify cause-effect relations between transitions (labels specify temporal intervals $[min, max]$, meaning that this transition must occur not before min and not after max temporal units). Also states are labeled to specify their durations.

Finally values of different state variables, activities that have to be scheduled over resources and cross relations between tasks and activities are linked each other with *synchronization* constraints (basically other labeled transition arcs that specify temporal relations between entities at their extremes). Cross relations allow the modeler to link causal theory and scheduling features: linking a task over a state variable with an activity over a resource we put scheduling into planning, allowing the modeler to require a resource usage during certain world's states, while from the other hand, linking an activity with a task over a state variable we put causal theory behind a scheduling problem, allowing the modeler to require some states of the world when an activity is scheduled.

This domain theory model is described using DDL.2 specifications. The DDL.2 domain description language (see (Cesta, Fratini, & Oddi 2004) for a more detailed description of this language) is an evolution with respect to the previous proposal called DDL.1 (Cesta & Oddi 1996).

As we have seen, state variables are basically finite timed state automata so the specification of a state variable involves the definition of its possible states and its allowed state transitions. In DDL.2 a state variable definition specifies its name and a list of values, called *states* that the state variable may take. Each state is specified with its name and a list of static variable types. Indeed the possible state variable values for DDL.2 are a discrete list of predicate instances like $\mathcal{P}(x_1, \ldots, x_m)$. For each state variable $\mathcal{SV}_i$ we specify: (1) A name that uniquely indicate this kind of component; (2) A domain $\mathcal{DV}_i$ of predicates $\mathcal{P}(x_1, \ldots, x_m)$ and (3) A domain $\mathcal{DX}_j$ for each static variable x_j in the predicate.

In Fig. 4(a) we show a possible DDL.2 model for the component $Satellite$ described in the last section where the type $TARGET$ takes values on a set containing labels for each interesting object in the world, plus, of course, a label for the Earth (while $target$ is an element of this domain). For each state we specify which states can follow it and which state can hold before it, expressing in that way which state transition rules are legal for that component. Value $Unlocked()$ for instance can hold at least for 1 second and there is not upper limit to the time the satellite can be in this idle status (statement $[1, +INF]$), and can be followed (Statement `MEETS`) by a value $Locking(target)$. Similarly `MET-BY` statement allows to specify which value the component can take just after $Unlocked()$ value. Roughly speaking this model describes a simple component which behavior is an alternation of sequences $\ldots Unlocked() \rightarrow Locking(target) \rightarrow Locked(target) \rightarrow Unlocked() \ldots$ and so on.

The main interesting feature of this language is the descrip-

```
SV Satellite (Unlocked(),Locking(TARGET),Locked(TARGET))
{
COMP
{
STATE Unlocked() [1, +INF] {      MEETS { Locking (x:TARGET); }
                                  MET-BY { Locked (y:TARGET);}}
STATE Locking(x) [1, +INF] {      MEETS { Locked (x)}
                                  MET-BY {(Unlocked())}}
STATE Locked(x) [1, +INF] {       MEETS { Unlocked ()}
                                  MET-BY { Locking (x)}}
}}
```

(a) DDL.2 State Variable description

```
COMP
{
STATE Perform_Observation (target,data) [time(target) , time(target)]

SYNC
{
DURING Satellite Locked(target) [5,+INF] [5,+INF];
BEFORE Instrument Download_Memory(data) [time(data) ,time(data)];
USE Memory memoryOcc(data) [0, 0] [0, 0] AFTEREND;
USE Channel memoryOcc(data) [0, 0] [0, 0] FROMSTARTTOEND;
}
}
```

(b) DDL.2 Compatibility

Figure 4: DDL.2 specifications

tion of the so called *compatibilities*, a sort of schema used to express synchronization constraints in a compact way. It should be easy at this point to understand how: every time that a task or an activity is added, compatibilities are used to generate some other task or activities that have to be scheduled over state variables or resources in order to follow the domain theory.

In our example some compatibilities have to be expressed for instrument components. When they take the value *Perform_Observation(target,data*) we need that (1) $target$ must be visible (then a synchronization is required with a $Locked(target)$ value for state variable *Satellite*; (2) Observed data have to be downloaded (then it must exists a following state *Download_Memory(Data)* in the same component behavior). It's worth nothing you can specify very complex constraints, like make data downloadable not before a certain slack of time (in order to perform some elaboration on it for instance) and not after another slack of time (in order to avoid information starving for instance); (3) You must have enough free memory and enough channel rate to perform your operations,then activities over *Memory* and *Channel* resource must be allocated. The amount of resource required from activities depends on how much data the instrument produce or it's able to transmit in time. Similarly for *Download_Memory* state some compatibilities have to be expressed, to be sure that only previous retrieved data can be downloaded and Earth is visible. In Fig. 4(b) there is a DDL.2 compatibility specification for a generic instrument[2].

Once you modeled the domain theory you can express the problem that have to be solved specifying some goals like actions that have to be performed by instruments components, as temporally situate *Perform_Observation* values, and/or a set of pre-defined activities that have to be allocated over resources in any feasible solution (goals over resources).

Starting from this problem specification we can build several networks of task or activity, one for each state variable or resource. These networks contains a logically feasible solution over a temporal network. Tasks and activities are linked each other via temporal constraints, and have to be scheduled to post more constraint until a solution is reached (meaning that each state variable assumes at least and no more than one value at each time instant and resources are never overused).

Affording the whole problem, starting from a domain theory description, is more useful than generate "blind" activities networks outside the architecture, then schedule them, because we can interleave expansion and scheduling steps, also changing tasks and activities in these networks if we realize we cannot schedule them (in fact DDL.2 allows the modeler to specify a disjunction of compatibilities that can be used to expand networks).

The solutions produced are feasible both from planning point of view (meaning that downloads follow observations and every operation is performed when the satellite is right oriented) and from the scheduling point of view (memory and channel are never overused).

Problem Solving

We have shown how in our framework we can dynamically generate networks of tasks or activities, where entities have to be allocated over state variables or resources. These entities share a common temporal model, and are linked to each other via temporal constraints. Thus we can reason on each of these networks, deducing, as typical in CSP problem solving, necessary constraints via propagation, and analyzing the underlying temporal problem in order to calculate which precedence constraints have to be posted to guarantee a feasible allocation of these entities over resources or state variables. All temporal constraints deduced via propagation and all precedence constraints posted as search decisions affect the whole shared temporal representation, and each decision, even about resource allocation or task ordering, has an automatic feedback over all other tasks and activities.

In the pure scheduling case methods to deduce *necessary* constraints by propagating informations about resource usages have been widely studied. It is possible do deduce a set of ordering constraints between activities that must be posted, just because otherwise a resource violation surely occurs. That is the case of the two activities shown in the top part of fig 5, which are supposed to be scheduled over a binary resource. From the underlying temporal network, we know that the first one can hold somewhere between t_1 (basically the lower bound of its starting point) and t_2 (the upper bound of its ending point), while the second one can hold between t_3 and t_4. Due to the fact that (1) They cannot overlap because the binary resource does not allow two activities to hold at the same time; (2) There is no way for the second activity to hold before the first one with respect to the temporal position of the involved start and end points; we can deduce that the second one must hold strictly after the first one. Thus a temporal constraints $[0, \infty]$ is automatically posted between the end of the first activity and the start of the second. The result is shown in the bottom part of the same figure.

[2] For the sake of brevity we do not show `MEET` and `MET-BY` statement. *time* and *memoryOcc* are two integer functions to calculate how many time need an observation and which amount of data it produce.

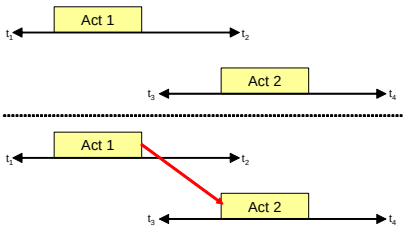


Figure 5: Propagation

Generally that is not enough in order to solve the problem. Sometimes a search decision is necessary. For instance in the left side of fig. 6 we show a situation with two activities that have to be scheduled again over a binary resource. But this time even analyzing the underlying temporal problem we are not able to calculate any necessary precedence constraint between these two activities: *both* orderings are temporally feasible. Thus a search decision step must be made, basically between the two feasible orderings shown in the right side of the same figure.

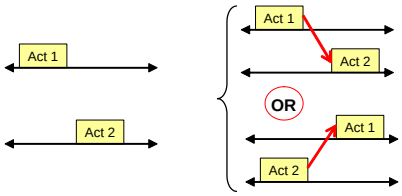


Figure 6: Scheduling decision

Of course constraints posted during the propagation step are necessary, i.e. they prune only non-feasible solutions, meaning that *any* feasible solution *must* contain these constraints. On the other hand scheduling precedence constraints are *search* decisions, thus they *could* cut some feasible solutions. Thus it could be necessary to backtrack during the search and choose a different ordering for some activities if it is not possible to find any solution from that point on.

In the case of more complicated types of resources than binary ones, like multi capacity resources, a *Precedence Constraint Posting* approach (Cesta, Oddi, & Smith 2002) can be applied to calculate a set of feasible ordering constraints between these activities that guarantee, once posted over the temporal network, that involved resources are never under or over used.

We go a little bit further, studying how to schedule tasks over state variables, basically following the guidelines we have shown above. Now we are going to show (1) a method for discovering an inconsistency of posted constraints (i.e. a temporal interval where the intersection of values allowed by constraints over that interval is an empty set) (2) a propagation algorithm which is ables to discover new necessary orderings between tasks in order to avoid necessary inconsistences over state variables.

Task networks scheduling

More formally with respect to what we did above, a task tk over a state variable SV can be thought as a tuple $\langle s, e, dur, S\rangle$ where s and e are the start and the end time points of an interval $[s, e]$ such that $e - s \leq dur$. If the state variable SV can take values over a set D, a task specifies a subset S of values in D that are allowed during the interval $[s, e]$. Roughly speaking a task specifies a disjunction of alternatives values from the domain D that the state variable can take during its temporal interval, as an activity specifies an amount of resources needed during its interval.

Follow an idea inspired from (Laborie 2003), for each task tk_i in the net we can compute, with respect to all the other tasks in the net, six different sets:

1. The set B_i of tasks that surely *end* before tk_i (s.t. $\forall tk_j \in B_i, e_j < s_i$);
2. The set A_i of tasks that surely *start* after tk_i (s.t. $\forall tk_j \in A_i, s_j > e_i$);
3. The set $O_i^<$ of tasks that surely *hold during the start point* of tk_i (s.t. $\forall tk_j \in O_i^<, s_j \leq s_i \wedge e_j > s_i$);
4. The set $O_i^>$ of tasks that surely *hold during the end point* of tk_i (s.t. $\forall tk_j \in O_i^<, s_j < e_i \wedge e_j \geq e_i$);
5. The set $U_i^<$ of tasks that are *unranked with respect to the start point* of tk_i, meaning that they can overlap tk_i or be ordered before tk_i, not after (s.t. $\forall tk_j \in U_i^<, s_j < e_i \wedge e_j$ is unranked with respect to s_i).
6. The set $U_i^>$ of tasks that are *unranked with respect to the end point* of tk_i, meaning that they can overlap tk_i or be ordered after tk_i, not before (s.t. $\forall tk_j \in U_i^>, e_j > s_i \wedge s_j$ is unranked with respect to e_i).

Roughly speaking we subdivide the set of tasks in three subsets: (1) Tasks that are no interesting, meaning that cannot create problems because they cannot overlap tk_i in any solution (sets B_i and A_i); (2) Tasks interesting for discover dead ends, because surely overlap tk_i in any solution (sets $O_i^<$ and $O_i^>$); (3) Tasks that can overlap or not tk_i (sets $U_i^<$ and $U_i^>$) that have to be analyzed to understand if it is possible to post some necessary precedence constraints. Basically the same situation showed in fig. 5.

We calculate for each task tk_i two sets:

$$I_i^{O<} = \bigcap_k S_k, \forall tk_k \in \{O_i^< \cup \{tk_i\}\}$$

$$I_i^{O>} = \bigcap_k S_k, \forall tk_k \in \{O_i^> \cup \{tk_i\}\}$$

basically the intersection of values allowed by surely overlapping tasks, respectively with respect to the start and the end point of tk_i.

Using these definitions we can easily discover dead ends: if $\exists tk_i \| (I_i^{O<} \equiv \emptyset \vee I_i^{O>} \equiv \emptyset)$, means it is not possible to find a set of values that the state variable can take when the task tk_i starts or ends because the intersection of all values allowed by the tasks in the net that surely overlap is empty. So there is not any feasible solution, meaning there is no way to order these tasks to avoid intervals where the state variable cannot take any value.

Similarly we calculate for each task tk_i two further sets:

$$I_i^{U<} = \bigcap_k S_k, \forall tk_k \in \{O_i^< \cup U_i^< \cup \{tk_i\}\}$$

$$I_i^{U>} = \bigcap_k S_k, \forall tk_k \in \{O_i^> \cup U_i^> \cup \{tk_i\}\}$$

basically the intersection of values allowed by both surely and unranked overlapping tasks, respectively with respect to the start and the end point of tk_i.

If we have $I_i^{O<} \neq \emptyset \wedge I_i^{U<} \equiv \emptyset$ or $I_i^{O>} \neq \emptyset \wedge I_i^{U>} \equiv \emptyset$, it means that not all tasks that *could* hold during the start point or the end point of tk_i can actually hold at the same time. This test is performed to understand if we can calculate some *necessary* ordering between tasks (obviously if all constraints that could overlap a given task can hold together no necessary constraints can be calculated). But if all tasks that could overlap tk_i cannot hold together (i.e. when $I_i^{U<} \equiv \emptyset$ or $I_i^{U>} \equiv \emptyset$), we investigate which of them cannot overlap tk_i. Then tk_i must be constrained to start after the end of each task $tk_j \in U_i^<$ such that its allowed values has empty intersection with values allowed by tk_i. When previous relation holds, a temporal precedence constraint between s_i and e_j must be posted. Symmetrically we can find winch tasks in $U_i^>$ must start after the end of tk_i. Roughly speaking we prune the search space calculating precedence constraints between pairs of tasks that could temporally overlap but that require sets of values for the state variable with empty intersection.

In general it is not possible to find a completely safe ordering for tasks through propagation only. Choices must be made between several order when two tasks can be ordered in two way (as in fig. 6), or when two task are not temporally related but, because their values, they cannot overlap for sure, or when there are groups of 3 or more tasks that exhibit pairwise non empty overlapping, but that cannot overlap all together. Following the same basic idea of Precedence Constraint Posting approach mentioned above (you have *peak* of resource contention when simultaneous activities require more than maximum amount of resource availability), in our case you have a *peak* of state variable contention when simultaneous tasks require sets of values with empty intersection. In fact we can schedule tasks as activities, analyzing "state variable profiles", finding *conflict sets* (a set of tasks potentially overlapping that have an empty intersection) and looking for feasible ordering that can solve the conflict.

The Architecture of OMP

OMP is an integrated constraint-based software architecture for planning and scheduling problem solving. This architecture implements an interval based planner and scheduler, built around the idea presented above. Basically, starting from a domain theory expressed in DDL.2 OMP builds tasks and activity networks over a shared temporal network, then schedules them as we explained.

The OMP software architecture essentially implements, from an abstract point of view, the typical CSP loop solving strategy, performing alternatively decision and propagation phases, starting from a CSP problem model. In an high level block view of OMP's architecture we see a *decision making* module that explores the search space posting constraints in a *constraint database*, that maintains information about the current partial solution and propagates decision effects pruning the search space, starting from a domain theory CSP model(see fig. 7).

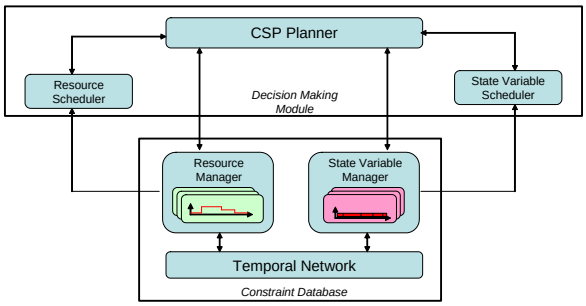


Figure 7: OMP Software Architecture

The temporal problem is managed and solved only via constraint propagation (this is because the STP is polynomially affordable) and, in this case, there is not a decision making phase, meaning that temporal constraints can be posted over a *temporal network* that calculates all necessary information (time lower and upper bounds of time points, plus distance between pairs of time points) using an All Pair Shortest Path algorithm.

Resource allocation and state variable management problems rather need both a constraint propagation phase and a solution search phase. Thus there are in OMP two modules, the *resource manager* and the *state variable manager*, that manage task and activity networks basically by performing constraints propagation strictly connected with two *scheduling modules* (in the decision making module) able to analyze the resource and state variable constraint databases and to calculate several sets of precedence constraints between activities and between tasks, precedence constraints that when posted over resource and state variable networks are able to guarantee that scheduling problems over these components are solved and each solution is feasible with respect to resource and state variable component constraints. These scheduling modules generate a search space, where at each node the decision making module can choose between different activity or task orderings. Resource propagation algorithms described in (Laborie 2003) have been implemented in a resource manager module, and the algorithm described below has been implemented for the state variable manager. Both for resource and state variable scheduling we basically use the precedence Constraint Posting Approach adapted to the features of this architecture.

Temporal, resource and state variable networks constitute, from a CSP point of view, the constraint database in the OMP software architecture. The decision maker module closes the loop. The *CSP planner* is the decision making module core: it explores the search space, making choices between: (1) which tasks and activities are necessary in order to force domain theory compliance and (2) which order to force among tasks on a state variable or activity over a resource when propagations are not able to prune all non-feasible orderings (this set is computed by the scheduler modules).

In fact the planner, starting from some goals (that is tasks

and activities that must appear in the final solution) dynamically unfolds the domain theory putting more tasks and activities into networks. Every time that a new task or activity is added we deduce, via propagation rules, new constraints that affect the situation of all networks, due to the shared temporal model. Moreover it is feasible to discover dead ends, and by interleaving scheduling and unfolding steps we integrate planning and scheduling (DDL.2 allows the user to model different expansions for a task or an activity, thus different networks can be built for the same problem).

Because OMP has been built starting from a scheduling core the primary aim of our preliminary experimentation has been to asses its scheduling capabilities. We have focused on the benchmark problem set described in (Kolisch, Schwindt, & Sprecher 1998). With a timeout of 100 seconds we were able to find at least one solution in more than 89% of the problems which are known to be solvable (with a 9.21% average deviation with respect to known best makespan) in the case of the *J30* problem set. We are currently focusing on modeling a set of meaningful integrated P&S domains which reflect real world application scenarios (see (Cesta, Fratini, & Oddi 2004) for an example).

Conclusions

This paper presented our current approach to planning and scheduling integration: we are studying how a constraint based scheduler can be endowed with the ability to synthesize new activities, i.e., by reasoning on planning knowledge.

We underlined how *Causal Knowledge* is the main distinguishing factor between planning and scheduling, thus building an architecture where causal reasoning can be performed behind time/resource reasoning allowed us to bridge the gap between these two AI research lines, extending from one hand pure planning schemes with quantitative time and resource reasoning and from the other hand extending pure scheduling schemes with a more complex domain theory.

Our aim was to solve integrated planning and scheduling problems by rationally integrating in the same architecture various of the state-of-the-art CSP algorithms. Then we showed as modeling the integrated planning and scheduling problem as concurrent evolving components allows us to afford it as network scheduling, where networks are automatically generated in the same architecture from a compact domain theory description and some goals that have to be achieved.

This integrated approach was built sharing a common CSP representation for all informations involved: time, resource and causal knowledge. Thus state of the art propagation and scheduling algorithms were used embedding them into the architecture. Thanks to our effort in exploiting similarities and differences between planning and scheduling we were able to present a framework where causal knowledge model and management are very close to a pure scheduling problem, then we were able to develop similar approaches for task scheduling.

References

Bartak, R. 2003. Visopt ShopFloor: Going Beyond Traditional Scheduling. In *Recent Advances in Constraints*, LNAI 2627, 185–199.

Cesta, A., and Oddi, A. 1996. DDL.1: A Formal Description of a Constraint Representation Language for Physical Domains. In M.Ghallab, M., and Milani, A., eds., *New Directions in AI Planning*. IOS Press.

Cesta, A.; Fratini, S.; and Oddi, A. 2004. Planning with Concurrency, Time and Resources: A CSP-Based Approach. In Vlahavas, I., and Vrakas, D., eds., *Intelligent Techniques for Planning*. Idea Group Publishing.

Cesta, A.; Oddi, A.; and Smith, S. F. 2002. A Constraint-based method for Project Scheduling with Time Windows. *Journal of Heuristics* 8(1):109–136.

Cheng, C., and Smith, S. 1994. Generating Feasible Schedules under Complex Metric Constraints. In *Proceedings 12th National Conference on AI (AAAI-94)*.

Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; Smith, B.; Fisher, F.; Barrett, T.; Stebbins, G.; and Tran, D. 2000. ASPEN - Automating Space Mission Operations using Automated Planning and Scheduling. In *Proceedings of SpaceOps 2000*.

Currie, K., and Tate, A. 1991. O-Plan: Control in the Open Planning Architecture. *Artificial Intelligence* 51:49–86.

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

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3-4):189–208.

Fox, M., and Long, D. 2003. PDDL 2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124. Special issue on 3rd International Planning Competition.

Frank, J., and Jonsson, A. 2003. Constraint based attribute and interval planning. *Journal of Constraints* 8(4):339–364. Special Issue on Planning.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL — The Planning Domain Definition Language, AIPS 98 Planning Competition Committee.

Jonsson, A.; Morris, P.; Muscettola, N.; Rajan, K.; and Smith, B. 2000. Planning in Interplanetary Space: Theory and Practice. In *Proceedings of the Fifth Int. Conf. on Artificial Intelligence Planning and Scheduling (AIPS-00)*.

Kolisch, R.; Schwindt, C.; and Sprecher, A. 1998. Benchmark Instances for Project Scheduling Problems. In J., W., ed., *Handbook on Recent Advances in Project Scheduling*. Kluwer.

Laborie, P., and Ghallab, M. 1995. Planning with Sharable Resource Constraints. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*.

Laborie, P. 2003. Algorithms for Propagating Resource Constraints in AI Planning and Scheduling: Existing Approaches and new Results. *Artificial Intelligence* 143:151–188.

Muscettola, N.; Smith, S.; Cesta, A.; and D'Aloisi, D. 1992. Coordinating Space Telescope Operations in an Integrated Planning and Scheduling Architecture. *IEEE Control Systems* 12(1):28–37.

Muscettola, N. 1994. HSTS: Integrating Planning and Scheduling. In Zweben, M. and Fox, M.S., ed., *Intelligent Scheduling*. Morgan Kauffmann.

Srivastava, B.; Kambhampati, S.; and Do, M. B. 2001. Planning the project management way: Efficient planning by effective integration of causal and resource reasoning in realplan. *Artificial Intelligence* 131(1-2):73–134.

Tsang, E. 1993. *Foundation of Constraint Satisfaction*. London and San Diego, CA: Academic Press.

Static Mapping of Hard Real-Time Applications Onto Multi-Processor Architectures Using Constraint Logic Programming*

Christophe Guettier
SAGEM, 178 rue de Paris
F-91300 Massy, France
Christophe.Guettier@sagem.com

Jean-François Hermant
INRIA, Rocquencourt, B.P. 105
F-78150 Le Chesnay, France
Jean-Francois.Hermant@inria.fr

Abstract

The increasing complexity of embedded real-time architectures, as well as the design of critical systems, has become both difficult to manage and costly. Mapping statically the set of tasks onto a multi-processor system is one of the most crucial issues. The designer must guarantee system feasibility, based on the system model and consideration of design decisions with respect to the requirement set. Our approach combines well-known online preemptive scheduling algorithms and their associated feasibility conditions, with problem solving techniques that are based on Constraint Logic Programming (CLP). We address a large class of online preemptive scheduling algorithms including so called fixed priority policies (Highest Priority First - HPF), as well as dynamic priority policies (specifically, Earliest Deadline First - EDF). The paper presents how to solve the mapping problem on representative examples, considering globally both task placement and hard real-time schedulability constraints. Optimization techniques are also experimented in order to minimize system dimensions. Lastly, we outline different recommendations for the design of efficient search strategies. Several benchmarks from various domains are considered as running examples throughout the paper.

Introduction

Deficiencies in the design or dimensioning of a critical and real-time high-performance system can cause fatal failures. In order to prove that for an entire system, real-time and architectural size requirements are met, it is necessary to prove its dimensioning. A relevant example of correct system dimensioning is ensuring that the processing resources are sufficient. This is a huge task as the increasing complexity of systems involves designs of combinatoric complexity. To master the complexity of system specification and design, we consider a proof-based methodology, like TRDF[1] as discussed in (Le Lann 97)(Le Lann 98). TRDF allows translation of the (incomplete and/or ambiguous) description of an application problem into a precise specification. To be acceptable, computer-based systems must come with proofs that all the decisions made during the system design satisfy system specification.

The complexity of target architectures has increased too, with many functional units incorporated on a single chip, the placement of tasks onto the set of processors remains a complex problem raising combinatorial explosions. It is necessary to consider simultaneously several interdependent subproblems such as task scheduling and placement. Usually highly combinatoric, each of these problems is characterized by feasibility constraints and design decisions (scheduling policy, data allocation strategy, communication protocols). Therefore, applying a proof-based method requires multiple formulations for the representation of the global system. The modelling phase of the method captures the invariants of the system specification, but also decomposes and simplifies its design complexity from coarse to fine grain. The designer has to solve a task placement problem, and must guarantee real-time schedulability. Evenmore, the designer needs to optimize the architecture to fit some space requirements or to optimize system performances.

Our approach relies on Constraint Logic Programming on finite parts of $\mathbb{N}$ CLP(FD) with all its classical operators (Van Hentenryck et al. 95). We claim that this language can solve and possibly optimize the design of complex hard real-time systems. One area of experimentation for Constraint Programming techniques has been the automatic data-layout of High-Performance FORTRAN programs onto parallel computers, using $0-1$ modelling, CPLEX and branch-and-bound searches (Bixby and al. 94). Using CLP, relevant results have also demonstrated the efficiency of the approach for modelling and solving Digital Signal Processing placement problems (Ancourt and al. 97),(Thiele 97). Another area of investigation has been the engineering of real-time (multi-processor) applications (Bacik and al. 98),(Guettier and Hermant 99). For all these examples, it has been necessary to solve the global problem globally using a composite model.

This paper presents a global approach for solving or optimizing automatically the mapping of real-time tasks onto multi-processor architectures using CLP. Focusing on the feasibility conditions of real-time scheduling algorithms, generic models and search methods are proposed to tackle the mapping feasibility, as well as the system size opti-

[1] TRDF is the French acronym of Real-Time Distributed Fault-Tolerant Computing.

mization. The different modelling steps required to express tractable feasibility conditions in CLP are detailed for an exhaustive class of online scheduling policies. It encompasses necessary and sufficient feasibility conditions for online scheduling algorithms such as Earliest Deadline First (EDF), Highest Priority First (HPF), as well as its specializations, Deadline Monotonic (HPF/DM), and Rate Monotonic (HPF/RM). In spite of their complexity, we demonstrate that analytical feasibility conditions can be expressed as equivalent feasibility constraints with CLP. Different composite search strategies are also proposed, combining load balancing and priority assignement (Audsley 1991) techniques. This association of offline solving techniques with online scheduling solutions is a new approach to the proof-based design of complex distributed and real-time systems.

Problem Specification

In this section, we specify the problem under consideration according to the TRDF method (Le Lann 97),(Le Lann 98). First, system requirement capture is summarised. Requirements are modeled in a fairly standard fashion, taking into account the problem models. Second, we state the major problem property, i.e. the timeliness property. As a result, it is possible to meet hard real-time constraints that would not be satisfied with a single processing element.

Models of System Requirements

In subsequent sections of the paper, demonstrations are performed using the assumptions that follow. System requirements define a global design problem that must be broken down into several sub-problems of lower complexity. Each sub-problem can be modeled with its own mathematical invariants. Finding a set of feasible solutions to a given sub-problem requires the instantiation of all the variables of the sub-problem. Relations resulting from this decomposition correlate model variables. They maintain the consistency of different local solutions and thus feasibility of the global solution.

Computational Model We assume the synchronous computational model (task durations have known bounds). Without loss of generality for the models described in this paper, we will use a discrete time model: $t \in \mathbb{N}$.

Task Model Let τ be the task set to execute. We consider a level of specification where each task $i \in \tau$ is represented by a sequence of steps, each step being mapped onto exactly one processor. A task has a worst case computation time $\forall i,\ C_i \in \mathbb{N}$ which can be constant or variable offline. When it is variable, upper and lower bounds exist and their values are known.

External Event Types Models All tasks are ready to execute whenever requested. Activation demands of a task are constrained by a periodic or sporadic arrival model (Mok 83).

Periodic	**Sporadic**
$\forall i, \exists \phi_{i,0} \in \mathbb{N},\ \forall a_i \in \mathbb{N},$	$\forall i, \forall j \in \mathbb{N},\ \exists \phi_{i,j} \in \mathbb{N},\ \forall a_i \in \mathbb{N}$
$d(a_i) = \phi_{i,0} + a_i T_i,$	$d(a_i) = \sum_{j=0}^{a_i} \phi_{i,j} + a_i T_i$
The period/minimal inter-arrival time of task i is denoted $T_i \in \mathbb{N}^*$	
The concrete or non-concrete attributes of task i:	
Concrete: $\phi_{i,0}$ known.	$\{\phi_{i,j}\}_{j\in\mathbb{N}}$ known.
Non-concrete: $\phi_{i,0}$ unknown.	$\{\phi_{i,j}\}_{j\in\mathbb{N}}$ unknown.
The $(a_i+1)^{th}$ activation date of task i is denoted $d(a_i)$.	
The first activation date of task i corresponds to $a_i = 0$.	

where $\phi_{i,0}$ is a phase difference/$\{\phi_{i,j}\}_{j\in\mathbb{N}}$ are phase differences. The sporadic model is more general than the periodic model (Mok 83) and our approach holds for both arrival models.

In subsequent sections of the paper, we consider a **periodic/sporadic non-concrete traffic** τ, which is a finite set of n periodic/sporadic non-concrete traffics τ_i. A periodic/sporadic non-concrete traffic τ_i captures an infinite set of periodic/sporadic concrete traffics ω_i. A periodic/sporadic concrete traffic ω_i is caracterized by its known activation dates $\{d(a_i)\}_{a_i\in\mathbb{N}}$, its worst case computation time C_i, its period/minimal inter-arrival time T_i, and its relative deadline D_i. $\forall a_i \in \mathbb{N},\ d(a_{i+1}) - d(a_i) \geq T_i$ and $0 \leq C_i \leq Min\{T_i, D_i\}$. The term traffic is commonly used in the telecommunication community to refer to a task set. These two terms can be used indifferently.

Task Placement Model For each task i, we use a coarsed grain placement form in order to size the task workload according to the whole target architecture. At a first glance, this can be formulated using classical set partitioning formulations.

Architectural Model The architectural model is MIMD "Multiple Instruction Flow, Multiple Data Flow", where the different processors can execute multiple instructions in parallel. The architecture is homogeneous, worst case computation time C_i does not depend on which processor is i is placed.

Property

To be feasible, the solution must satisfy the timeliness property, under the models described above.

Tasks are assigned timeliness constraints: latest termination deadline. A solution is said to be feasible if for every possible system run, every timeliness constraint is met. Values of deadlines are dictated by application considerations. A deadline can be expressed as a natural number: $\forall i \in \tau,\ D_i \in \mathbb{N}^*$

Example of real world specifications

This section presents three realistic specifications extracted from real world applications discussed in (Tilman et al. 01),(Guettier and al. 01), that are used as on-going examples throughout this paper. Althought various cost functions could be investigated, finding a feasible solution on a minimal set of processors is considered to be the engineering cost objective.

Task	$C(ms)$	$T(ms)$	$D(ms)$
insert_target	50	250	100
distance_eval	100	500	150
pursuit_target	150	500	300
suppress_target	20	200	500

Figure 1: Example of a detection system.

Task	$C(ms)$	$T(ms)$	$D(ms)$
FDIR	20	100	100
energy_manager	100	500	400
camera_controler	40	100	100
mmemory_controler	400	600	1000
telecom_protocol	100	400	200
antena_controler	40	100	200
unload_protocol	200	400	200

Figure 2: Example of an observation spacecraft *system.*

Detection system: As a simpler example, we propose a detection system (typical of airborne radars or sonars) that manages a list of targets. A Digital Signal Processing system (out of the scope of the example) processes a beamforming algorithm (Ancourt and al. 97).

Spacecraft system: This system performs some observations and saves its data on a mass memory (Tilman et al. 01). Controled remotely from earth, the spacecraft can nevertheless perform Fault Detection Isolation and Recovery. Data can be unloaded or new pictures ordered using a communication antenae.

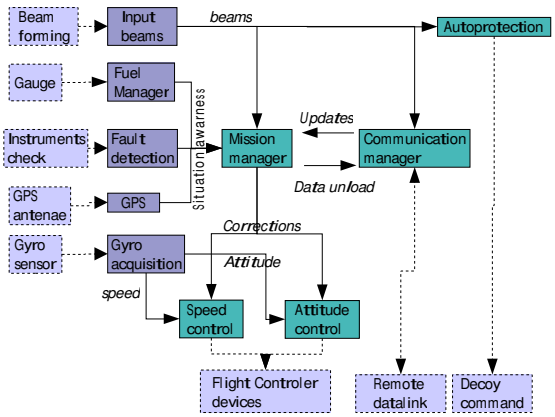
UAV Avionic system: It is a typical (although very simplified) core functional architecture (fig. 3) of a Unmanned Aerial Vehicle (UAV) system (Guettier and al. 01). It performs some observations that are downloaded, following a predefined mission plan. According to situation awarness, the mission manager task sends limited corrections to the speed and altitude controller tasks. It can also send situation information to a remote operator.

Hard real-time scheduling & mapping models

This section presents the constraint based model of the hard real-time scheduling and mapping problem. Feasibility constraints of scheduling algorithms are extracted from the state of the art in hard real time computing and modeled using a CLP(FD) language.

The mapping model, based on a set partitioning formulation is at first defined. A more generic formulation than well-known uniprocessor scheduling feasibility conditions is then detailed. These necessary and sufficient constraints are involving variables C_i, T_i, D_i as well as task partitioning. The modelling includes Highest Priority First (HPF) like Deadline Monotonic (DM), Rate Monotonic (RM) as well as Dynamic Priority classes of scheduling such as Earliest Deadline First (EDF).

In order to improve the problem solving efficiency, several decisions in problem representation have to be made. In the sequel, we explain how the modelling is refined in order to extract sufficient conditions, used as heuristic constraints.



Task	$C(ms)$	$T(ms)$	$D(ms)$
attitude_control	3	8	8
fuel_manager	2	30	20
mission_manager	4	20	11
gps_update	5	40	11
autoprotection	6	40	16
fault_detection	5	20	11
speed_controler	2	10	11
gyro_acquisition	6	10	15
beam_input	4	10	15
com_manager	2	15	14

Figure 3: Example of an UAV avionic *system.*

Furthermore, throughout the modelling, constraints are refined in order to be tractable with CLP capabilities.

Mapping models

The mapping of real-time applications can be decomposed into task placement, architectural and relational models. Set partitioning is expressed using $0-1$ formulation and represents task distribution. The architectural model is then expressed as a resource constraint. Additional relational statements ensure the global solution consistency.

Task placement as set partitioning We use a $0-1$ formulation to specify whether the processor p is allocated to a given task i. This model has been widely investigated for the representation of various combinatorial problems involving set partitioning (Gondran and Minoux 95).

$$\forall i \in \tau,\ \forall p \in [0, \mathbf{P}_{max}),\ m_p^i \in \{0,1\}$$
$$\forall i \in \tau,\ \sum_{p=0}^{\mathbf{P}_{max}-1} m_p^i = 1 \qquad (1)$$

Statement (1) specifies that a task i is allocated to a single processor p iff $m_p^i = 1$. Using this formulation, the actual partitioning may lead to a lower number of busy processors.

Architectural resource constraints The number of busy processors $\mathbb{P}$ has an upper-bound determined by the constant $\mathbf{P}_{max}$. It defines the number of processors required to map the entire task and is given by $\mathbb{P} = card\{p \in [0, \mathbf{P}_{max}) | \exists i \in \tau / m_p^i = 1\}$.

$$\mathbb{P} = \sum_{p=0}^{\mathbf{P}_{max}-1} \underset{i \in \tau}{max}\{m_p^i\} \quad (2)$$

Relational Constraints Relational statements between models are required to retrieve global consistent solutions and to globally optimize cost functions such as the system size.

$$\forall i \in \tau,\ \forall p \in [0, \mathbf{P}_{max}),\ C_i(p) = C_i.m_p^i \quad (3)$$

Modelling feasibility constraints of online scheduling algorithms

The scheduling algorithms under consideration belong to the class of online real-time scheduling algorithms. In this class, there are two subclasses: that of deadline-driven scheduling algorithms and that of fixed-priority scheduling algorithms. EDF belongs to the former subclass and dominates any other scheduling algorithm belonging to the latter subclass, such as, for instance, Highest Priority First/Deadline Monotonic (HPF/DM) and Highest Priority First/Rate Monotonic (HPF/RM) (Dertouzos 1974).

This section gives feasibility conditions for EDF and HPF policies, which allows us to conduct a comparative study in terms of constraints complexity.

Basic concepts This section defines the basic concepts introduced in (Liu and Layland 73). Further sections extend the approach and recast feasibility conditions into tractable constraints for CLP languages.

The workload $W(p,t,\tau)$: By definition, the workload $W(p,t,\tau)$ for each processor p is the amount of time that is needed to run all the tasks whose activation times are in $[0,t)$ (Baruah et al. 1990b). To give the expression of $W(p,t,\tau)$, we consider the synchronous concrete traffic $\omega \in \tau$, where $\forall i, \forall j \in \mathbb{N},\ \phi_{i,j} = 0$.

$$W(p,t,\tau) = \sum_{j \in \tau} W(p,t,j) = \sum_{j \in \tau} \left\lceil \frac{t}{T_j} \right\rceil C_j(p). \quad (4)$$

A necessary feasibility condition (NC): This well-known necessary condition can be used as a heuristic to solve the global problem.

$$\tau \text{ is feasible by EDF, HPF} \Rightarrow \forall p \in [0, \mathbf{P}_{max}), \sum_{j=1}^{n} \frac{C_j(p)}{T_j} \leq 1. \quad (5)$$

sketch of the proof:

We derive the utilization factor $U(p,\tau)$ from the workload $W(p,t;\tau)$:

$$U(p,\tau) = Lim_{t \to \infty} \left\{ \frac{W(p,t;\mathcal{P}_p(\tau))}{t} \right\} = \sum_{j=1}^{n} \frac{C_j(p)}{T_j}.$$

By definition, the utilization factor $U(p,\tau)$ for processor p is the fraction of time that is needed to run all the tasks over $[0,\infty)$, i.e., the limit of $W(p,t;\tau)/t$ as t tends to infinity. If $\mathcal{P}_p(\tau)$ is feasible by EDF, then $U(p,\tau) \leq 100\%$.

Using exact feasibility conditions for EDF

The EDF policy works as follows (Liu and Layland 73). At any time $t \in \mathbb{R}^+$, if there are pending tasks (i.e., tasks which have been previously activated but which have not been fully completed yet), EDF runs the task which has the earliest absolute deadline. The processor is then said to be busy. To decide between tasks having the same absolute deadline, EDF makes use of a tie-breaking rule (e.g., a arbitrary order). If there are no pending tasks, EDF runs no task. The processor is then said to be idle (see example in figure 4). In this paper, we consider preemptive versions for EDF and its associated analytical feasibility conditions.

The processor demand $h(p,t,\tau)$:

$$h(p,t,\tau) = \sum_{j=1}^{n} h(p,t,j)$$

$$= \sum_{j \in \tau}^{n} Max\left\{0, 1 + \left\lfloor \frac{t - D_j}{T_j} \right\rfloor \right\} C_j(p) \quad (6)$$

By definition, the processor p demand $h(p,t,\tau)$ is the amount of time that is needed to run all the tasks whose activation times and absolute deadlines are in $[0,t]$ (Baruah et al. 1990b). To give the expression of $h(p,t,\tau)$, we consider the synchronous concrete traffic $\omega \in \tau$.

A necessary and sufficient feasibility condition (NSC):

$$\tau \text{ is feasible by EDF} \Leftrightarrow$$
$$\forall t \in \mathbb{R}^+,\ h(p,t,\tau) \leq t; \quad (7)$$

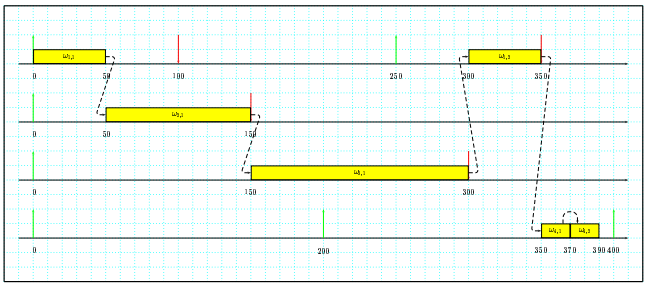
$$\tau \text{ is feasible by EDF} \Leftrightarrow Sup_{t \in \mathbb{R}^{+*}} \left\{ \frac{h(p,t,\tau)}{t} \right\} \leq 1. \quad (8)$$

sketch of the proof:

By definition, the processor demand $h(p,t,\tau)$ is the amount of time that is needed to run all the tasks whose activation times and absolute deadlines are in $[0,t]$. τ is feasible by EDF, if and only if, $\forall p \in [0, \mathbf{P}_{max}), \forall t \in \mathbb{R}^+,\ h(p,t,\tau) \leq t$, i.e., if and only if, $Sup_{t \in \mathbb{R}^{+*}} \{h(p,t,\tau)/t\} \leq 100\%$.

Study interval: In order to be operationally satisfied, feasibility constraints cannot be stated on $\mathbb{R}^{+*}$. Although the processor busy period λ (Hermant et al. 96),(Hermant 98) is a candidate interval, the resulting constraints are not tractable in a CLP model. Instead, the study interval is set to $L = \underset{i \in \tau}{gcm}(T_i)$ and is preprocessed. Therefore, equation (8) becomes:

$$\tau \text{ is feasible by EDF} \Leftrightarrow Sup_{t \in (0,L)} \left\{ \frac{h(p,t,\tau)}{t} \right\} \leq 1 \quad (9)$$



$$U(0,\tau) = \sum_{i=1}^{n} \frac{C_i}{T_i} = 0.8 \quad (10)$$

$$\lambda = \sum_{i=1}^{n} \left\lceil \frac{\lambda}{T_i} \right\rceil C_i = 390ms \quad (11)$$

$$\forall t \in [0, 390),\ h(0,t,\tau) \leq t \quad (12)$$

Figure 4: Mapping of the `detection system` onto a single processor

Example of feasible mapping: A feasible mapping of the `detection system` onto a single processor can be derived. The utilization factor $U(0,\tau)$ is given in (10). Instead of L, the length of the busy period λ is used as the study interval (11). Lastly, the necessary and sufficient feasibility condition for EDF (12) holds true, illustrated by a feasible schedule of the synchronous activation scenario (worst cases) in fig. 4.

Using exact feasibility constraints for HPF At any time $t \in \mathbb{R}^+$, if there are pending tasks (i.e., tasks which have been previously activated but which have not been fully completed yet), HPF runs the task which has the highest priority (Liu and Layland 73). Priorities can be allocated according to deadlines (said to be deadline monotonic) or period (also called rate monotonic). The highest priority correponds respectively to the lower deadline or the lower period. Priorities can also be computed statically in order to optimize the execution (Audsley 1991), for example, by maximizing the workload. As for EDF, we consider a periodic/sporadic non-concrete traffic τ, which is a set of n periodic/sporadic non-concrete traffics τ_i. When a set of tasks is allocated to the same processor, a priority order is associated to τ, as follows.

Priorities:

$$\forall i \in \tau, \forall j \in \tau / i \neq j,\ \exists p\ /$$
$$(m_i^p = 1) \wedge (m_j^p = 1) \Leftrightarrow\ (j \prec i) \nabla (i \prec j) \quad (13)$$

where ∇ the exclusive or.

Response time $r(p,t,i,\tau)$**:** To establish necessary and sufficient conditions as schedulability constraints for HPF (Lehoczky 90), (Tindell and al. 94), one has to consider the worst case response time $r(p,t,i,\tau) \in \mathbb{N}$ associated to task i and processor p. The task set is feasible, if and only if, for each task i, at any activation t, the response time meets the deadline D_i:

$$\tau \text{ feasible by HPF} \Leftrightarrow$$
$$\forall p \in [0, \mathbf{P}_{max}), \forall i \in \tau, \forall t \in [0,\infty), r(p,t,i,\tau) \leq D_i$$

Workload $w(p,t,i,\tau)$**:** For each task, the response time can be rewritten using the workload $w(p,a,i,\tau) \in \mathbb{N}$, as follows:

$$\forall i \in \tau, \forall p \in [0, \mathbf{P}_{max}), \forall t \in [0,\infty),$$
$$r(p,t,i,\tau) = w(p,t,i,\tau) - t$$

Let us consider a task i (of priority i). In $[0,t]$, the maximum number of executions of task i is $1 + \lfloor t/T_i \rfloor$. For each task j (of lower priority $j \prec i$), in $[0, w(p,t,i,\tau))$, the maximum number of executions of task j is $\lceil w(p,t,i,\tau)/T_j \rceil$. Hence, the workload can be written as follows:

$$w(p,t,i,\tau) = \left(1 + \left\lfloor \frac{t}{T_i} \right\rfloor\right) C_i + \sum_{j \in \tau / j \prec i} \left\lceil \frac{w(p,t,i,\tau)}{T_j} \right\rceil C_j(p) \quad (14)$$

The resulting necessary and sufficient condition: By replacing $r(p,t,i,\tau)$, it follows that:

$$\tau \text{ feasible by HPF} \Leftrightarrow \forall i \in \tau, \forall t \in [0,\infty), \forall p \in [0, \mathbf{P}_{max}),$$
$$w(p,t,i,\tau) \leq t + D_i \quad (15)$$

This set of equations converges to a fixed point, which can be solved using the fixed point semantics of CLP languages. A tractable interval for activations $[0,t)$ must be specified for implementing the constraint using a CLP language. First, a maximal study interval can be specified using the greater common multiple of the periods/sporadicities. Second, for each task, activations can be formulated using the associated period/sporadicity according to the worst case analysis. This leads to:

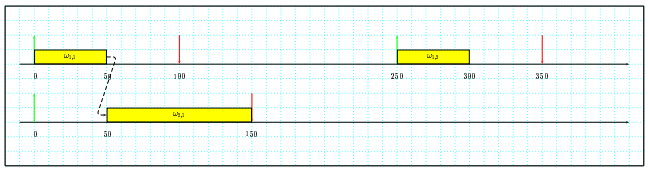
$$\forall i \in \tau,\ t \in [0, \underset{j \in \tau}{lcm}(T_j))$$
$$\forall i \in \tau, \forall t \in [0, \underset{j \in \tau}{lcm}(T_j)), \exists q_i \mid t = q_i.T_i \Rightarrow \quad (16)$$
$$\forall i \in \tau, \forall q_i \in [0, \underset{j \in \tau}{lcm}(T_j)/T_i) \quad (17)$$

As for EDF (in eq. 9), the lcm constraint, which has not received efficient implementation yet, is preprocessed. Here again, upper-approximation may be found in order to keep the global problem tractable by CLP. Using constraints (16,17) constraint (14) can be simplified:

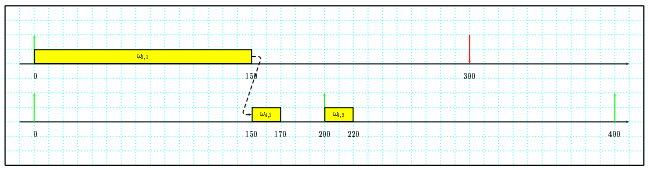
$$w(p,q_i,i,\tau) = (1+q_i)C_i + \sum_{j \in \tau / j \prec i} \left\lceil \frac{w(p,q_i,i,\tau)}{T_j} \right\rceil C_j(p) \quad (18)$$

such that the NSC (15) becomes:

Processor 0:



Processor 1:



$$U(t, 0, \tau) = U(t, 1, \tau) = 40 \quad (21)$$

$$\lambda_0 = 150 \text{ ms and } \lambda_1 = 170 \text{ ms} \quad (22)$$

Figure 5: Mapping of the `detection_system` using HPF/DM or HPF/RM and HPF/DM onto two processors.

$$\tau \text{ feasible by HPF} \Leftrightarrow$$
$$\forall i \in \tau, \forall q_i \in [0, \underset{j \in \tau}{lcm}(T_j)[, \forall p \in [0, \mathbf{P}_{max}),$$
$$w(p, q_i, i, \tau) \leq q_i.T_i + D_i \quad (19)$$

Lastly, the priority order (13) may not be instantiated. Therefore, the formulation (18) leads to the enumeration of all priority order without the consideration of constraint propagation. A more efficient approach is to model the priority order using a permutation matrix P, such that statement (18) $\Leftrightarrow$ (20):

$$P \in \{0,1\}^{card(\tau) \times card(\tau)}$$
$$\vec{C'} = P.\vec{C},\ \vec{T'} = P.\vec{T},\ \vec{w'}(p, q_i, \tau) = P.\vec{w}(p, q_i, \tau)$$
$$w'(p, q_i, i, \tau) = (1 + q_i)C'_i + \sum_{j \in [0,i)} \left\lceil \frac{w'(p, q_i, i, \tau)}{T'_j} \right\rceil C'_j(p) \quad (20)$$

Example of feasible mapping: An optimal mapping (in terms of the number of processors) of the `detection system` can be given on two processors, using a HPF policy (fig. 5). It considers tasks `insert_target` and `distance_eval` placed onto processor 0, while tasks `pursuit_target` and `suppress_target` are placed onto processor 1. For processor 0, task `insert_target` is assigned the highest priority, while `task distance_eval` is assigned the lowest. This assignment corresponds to the Deadline Monotonic (DM) or the Rate Monotonic (RM) fixed priority assignment. For processor 1, task `pursuit_target` is assigned the highest priority, while task `suppres_target` is assigned the lowest. This assignment corresponds to the Deadline Monotonic fixed priority assignment.

This placement defined can also be computed automatically using the CLP models implementation.

Automatic Solving Using CLP Language

Several problems can be solved using models presented. Specific periods, deadlines, response or computation times that would be related to an applicative function (such that the worst case durations of the UAV control tasks) can be solved. However, within the scope of this paper, the focus is on a non-functional requirement: the minimal number of required processors $\mathbb{P}$.

Search strategies

This section presents the main design concepts for search strategies dedicated to the scheduling and mapping problem under consideration.

Elementary search The elementary search strategy is based on the `labeling` and Branch & Bound predicates, provided in most of CLP implementations (Carlsson et al. 97). Even when problem models are tractable, these basic strategies are too weak to cope with the largest problem instances.

Load balancing strategies So called load balancing heuristics can be introduced as static heuristics, or as a dynamic search strategy. These heuristics and strategies can be used for both HPF and EDF scheduling policies.

Static heuristic: This strategy is similar to elementary search, but disjunctive constraints on the utilization factor are added in order to statically decompose the search space. This strategy structures the search space in favour of assignements that do not under-utilize processors (also called no starvation heuristic).

Dynamic balancing search strategy: Instead of using standard `labeling` predicates, this strategy reorders dynamically[2] the set of variables to explore. Each time a placement variable m_q^j is successfully instantiated (which indicates that task j is placed on processor q), a set of variables $M_p = \{m_p^1 \ldots m_p^n\}$ is selected according to the least utilized processor p. The selection function tests the minimal bounds of constraint variables $\{U(k, \tau)\}_0^{\mathbf{P}_{max}}$. Then, the strategy attempts to instantiate one variable of the set M_p. If one variable can be successfully instantiated, the strategy starts over until all the tasks are placed. Otherwise, it backtracks and another processor is selected according to the increasing order of utilization factors.

Uniprocessor optimal priority assignment heuristic: The last search strategy to be investigated relies on the uniprocessor optimal priority assignment (PA) (Audsley 1991). Using this method, only the problems involving HPF scheduling policies can be improved. Far from being optimal on multi-processor problems, this PA method can nevertheless be used as a heuristic. The search strategy first

[2]Here, the term 'dynamic' does not mean that the strategy is performed online, but that the construction of the search tree is dynamic (e.g. during the search itself).

enumerates the set of placement variables, for each processor and each task m_p^i. Then, for each processor, the optimal PA is performed. It consists of instantiating tasks of a highest priority first, without backtracking on these assignments (Audsley 1991). Still complete, this technique obviously simplifies the search complexity. The PA heuristic can be used in combination with both static and dynamic balancing strategies but may conflicts with the load balancing strategies.

Experiments

Six combinations of strategies (figure 6) and scheduling policies have been experimented on 522 problem instances, optimising for the number of processors. The elementary strategy without any heuristic does not give meaningful results in reasonable time, until the static load heuristic is added.

Problem generation method

The experimentation method relies on the real world examples. A problem generation technique operates by iteratively decreasing of 5% the task deadlines until problem instances becomes infeasible for the maximum number of processors. This way, problem instances become more difficult by decreasing the global laxity of the hard real-time constraints. Experiments are performed on a 1Ghz Intel III processor, with 256 MBytes of main memory and WindowsXP Pro and SICStus Prolog 3.9.0, using finite domain constraint library (Carlsson et al. 97).

Global results

The subset of experiments under consideration (figure 6) enables the comparison of the different policies (namely EDF and HPF) for the mapping problem, as well as the various strategies proposed. Figure (figure 6) includes data gathered from the three benchmarks. On this figure, the column *best solutions* is the number of instances for which at least one solution is retrieved. The column *proof of completeness* is the number of instances for which the problem is solved optimally or for which it can be proved that no feasible solution exist.

policy, strategy	number of instances	best solution	proofs of completeness
EDF, static load	87	82	62
HPF, static load	87	65	26
HPF, static load, PA	87	76	39
EDF, dynamic load	87	86	66
HPF, dynamic load	87	66	26
HPF, dynamic load, PA	87	57	41

Figure 6: Experiments under consideration

At first glance, experiments suggest that solving the mapping problem with an EDF scheduling policy is easier than with its HPF equivalent. Globally, the priority assignment heuristic improves the search efficiency (for HPF cases), when the static load heuristic is activated. For HPF cases, and without the priority assignment heuristic, the strategies fail to prove completeness for any of the problem instances in benchmark `spacecraft_system` and `uav_avionic` (completeness can be proved only for the 26 easy instances of the `detection_system` benchmark).

When considering the EDF policy, the number of problems solved is 5% better (6% better for the proofs of completeness) by replacing the static load balancing heuristic with a dynamic search. This is also the case for the HPF policy using the priority assignments heuristic, but only for the proof of completeness (41 versus 39 instances). As a result of the conflicting strategies, the dynamic load strategy combined with the PA heuristic gives a reduced performance on the number of problems solved (57 versus 66 instances).

Evaluation of solutions

Figure 7 compares the number of processors found for the two non-trivial benchmarks (`spacecraft_system` and `uav_avionic`), when decreasing the global laxity. Using the EDF policy, search strategies find a smaller number of processors. Furthermore, using this policy, more problems can be (optimally) solved in the time limit imposed. By convention, when the strategy fails to give any feasible solution or to prove that the problem instance is not feasible, a vertical line is drawn. As for the global results, the priority assignment heuristic improves the costs, except for the conflicting load balancing strategy.

Strategy performances

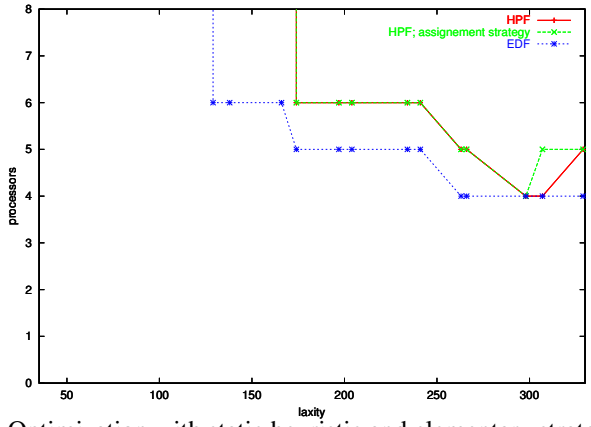
For the purpose of the experiments, a time-out is imposed on the different runs. Therefore, experiments that successfully yield to a proof of optimality (mainly on benchmarks `detection_system` and `spacecraft_system`) are separated from experiments that lead to optimized solutions, without guarantee of optimality (mainly on benchmarks `spacecraft_system` and `uav_avionic`).

Time for proving optimality Figure 8 gives the different completion times for benchmark `spacecraft_system` to prove solution optimality according to the different search strategies. Concerning the benchmark `detection_system`, the proof of optimality can be performed for all the experiments with 2 and 4 processors. However, due to the little differences between solving duration and the small problem size, the curves are not represented in this paper.

With the EDF policy, proof of completeness can be performed on all the problem instances of the `spacecraft_system` in figure 8, and similarly on parts of the problem instances of the `uav_avionic`. In general, the time for proving the optimality is not modified when replacing the load balancing heuristic with the dynamic load balancing strategy. For the benchmark `spacecraft_system`, and using the priority assignment heuristic, proofs of optimality can be performed for a subset of instances. The dynamic load balancing strategy improves the solving time only for two instances. Without the priority assignement heuristic, strategies fail to prove completeness for the HPF scheduling policy in the time limit (7 seconds).

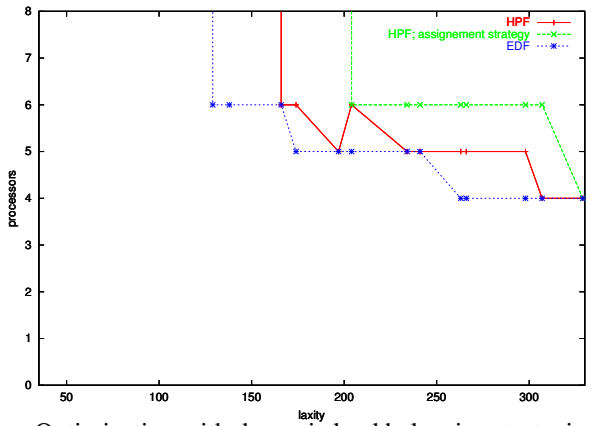
Time for finding an optimized solution Figure 9 gives the completion times for benchmark `uav_avionic` to find

a.



Optimization with static heuristic and elementary strategy

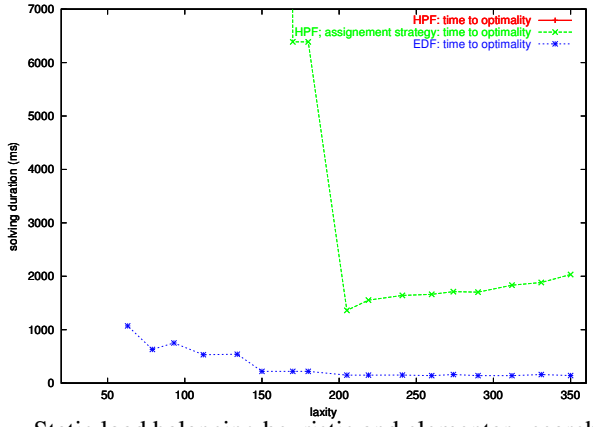
b.



Optimization with dynamic load balancing strategies

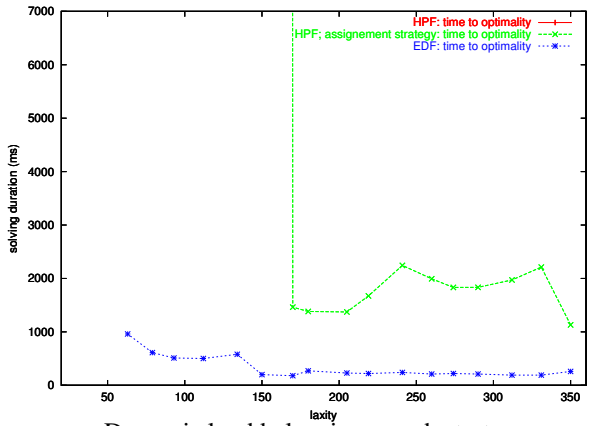
Figure 7: The `uav_avionic` example (6 processors): comparison of the minimal number of processors with different search strategies

a.



Static load balancing heuristic and elementary search

b.



Dynamic load balancing search strategy

Figure 8: Time to complete proof of optimality on the `spacecraft_system` example (4 processors)

an optimized solution. The time out range is 20 seconds. Again by convention, a vertical line is drawn when the solving strategy fails to produce any feasible solution or to prove that the problem instance is not feasible. It is not obvious that the dynamic load balancing strategy improves the solving duration on 9-a. Furthermore, as shown on figure 9-b, the conflict of this strategy with the priority assignment heuristic gives a counter-performance (instances under 200 of laxity cannot be solved). In contrast, for the EDF policy, strategies are improved with the load balancing heuristic, even for the set of non-feasible instances (as shown on figures 9-a and 9-b).

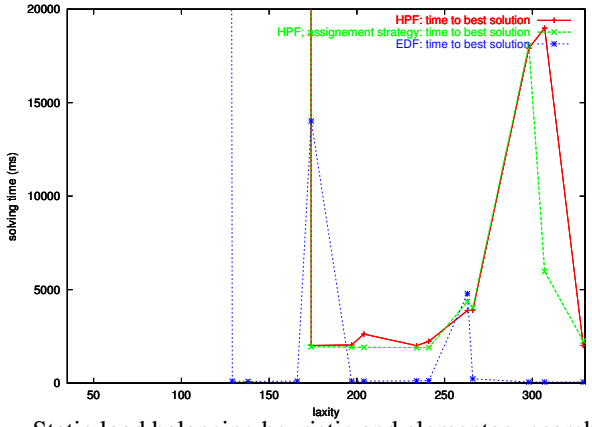
Conclusions and Further Work

This paper demonstrates that the necessary and sufficient conditions can be preserved while modelling the feasibility of HPF and EDF preemptive scheduling policies using a CLP language. From a Constraint Programming point of view, this represents an interesting alternative to the fully static approach generally considered, where preemptive scheduling is solved completely offline. The constraint-based modelling described in this paper has a larger applicative impact, as many real-time operating systems for embedded applications make use of HPF policies. Furthermore, using these constraint-based models, various mapping problems can be solved. On realistic benchmarks, experiments have shown that proofs of completeness can be found in reasonable time.

The paper illustrates how to support the engineering of distributed real-time systems using a proof-based method. As a matter of fact, a generic heuristic, such as load balancing performs better with EDF. With other HPF policies, the benefit of this heuristic is not clear, even using dynamic search techniques that are generally recognized as strong strategies. It is very difficult to conclude on the gain of the priority assignment heuristic, although dedicated to the HPF scheduling policy.

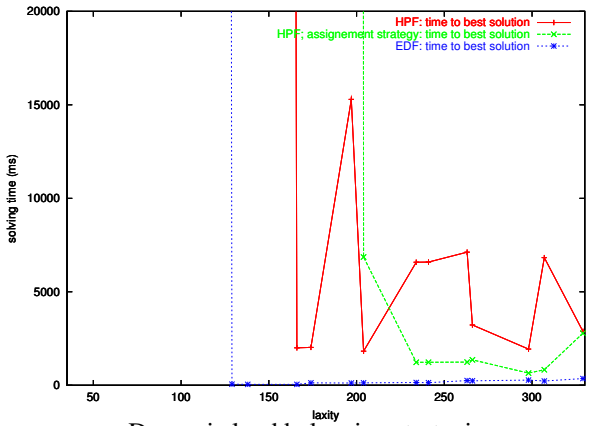
Most importantly, the paper shows for real world examples that the design of distributed real-time architectures is simpler using EDF policies. On various problem instances with the EDF policies, strategies have been able to prove search completeness. As a result, the solution optimality

a.



Static load balancing heuristic and elementary search

b.



Dynamic load balancing strategies;

Figure 9: The `uav_avionic` example (6 processors): time to retrieve optimized solutions using the different strategies

or the non feasibility of these problem instances can be decided, proven and guaranteed. For the same instances, strategies fail to prove search completeness using HPF policies. This paper is a first step towards the promising combination of offline and online multi-processor scheduling techniques. The modelling of communication protocols shall be part of further works.

References

N. C. Audsley, *Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times*, Tech. Report YCS 164, Dept. of Computer Science, University of York, December 1991.

S. Baruah, A. Mok, L. Rosier, *Preemptively scheduling hard real-time sporadic tasks on one processor*, 11th Real-Time Systems Symposium, pp. 182-190, 1990.

M. Dertouzos, *Control Robotics: the procedural control of physical processors*, Proceedings of the IFIP congress, pp. 807-813, 1974.

J.-F. Hermant, *Analysis of Real-Time Distributed Scheduling Algorithms*, PhD Thesis, University of Paris VI, 1999.

J.-F. Hermant, L. Leboucher, N. Rivierre, *Real-time fixed and dynamic priority driven scheduling algorithms: theory and experience*, INRIA Research Report 3081, 142 p., 1996.

J. P. Lehoczky, *Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines*, 11th IEEE Real-Time Systems Symposium, pp. 201-209, Dec. 1990.

G. Le Lann, *Proof-Based System Engineering for Computing Systems*, 1^{st} Joint ESA/INCOSE Conference on Systems Engineering - The Future, IEE/ESA Pub., Vol. WPP-130, 5a.4.1-5a.4.8., Nov. 11-13, 1997.

G. Le Lann, *Proof-Based System Engineering and Embedded Systems*, School on Embedded Systems, Veldhoven (NL), Nov. 1996, Lecture Notes in Computer Science, vol. 1494, Springer Verlag Pub., pp. 208-248, 1998.

C.L. Liu, J.W. Layland, *Scheduling algorithms for multiprogramming in a hard real-time environment*, Journal of the Association for Computing Machinery, 20(1), pp. 40-61, 1973.

A.K. Mok, *Fundamental design problems for the hard real-time environments*, PhD, MIT/LCS/TR-297, 1983.

K. W. Tindell, A. Burns, A. J. Wellings, *An extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks*, The Journal of Real-Time Systems, Vol. 6(2), pp. 133-152, Klüwer Academic Publishers, March 1994.

C. Ancourt, D. Barthou, C. Guettier, F. Irigoin, B. Jeannet, J. Jourdan, and J. Mattioli, *Automatic mapping of signal processing applications onto parallel computers*, In Proc. ASAP 97, Zurich, July, 1997.

R. Bixby, K. Kennedy, and U. Kremer, *Automatic Data Layout Using 0-1 Integer Programming*, In Proc. PACT94, Montreal, Canada, August, 1994.

A. Bakic, M. W. Mutka, D. T. Rover, *Using Constraint Logic Programming for Engineering of Real-Time Systems*, Michigan University Technical Report, MSU-CPS-98-23, 1998.

C. Guettier, J.-F. Hermant *A Constraint-Based Model for High-Performance Real-Time Computing*, In Proc. of the Intl. Conf. on Distributed and Parallel Systems, Florida, September, 1999.

C. Guettier, B. Patin and J.-F. Tilman *Validation of Autonomous Concepts using the ATHENA environment*, In Proc. of the European Space Technology and rEsearch Center (ESTEC) Workshop On-board Autonomy, October, 2001.

U. Kremer, Optimal and Near-Optimal *Solutions For Hard Compilation Problems*, Parallel Processing Letters 7(4), World Scientific Publishing Company, 1997.

L. Thiele, *Partioning Processor Arrays under Resource Constraints*, In Journal of VLSI Signal Processing Systems, 15, pp. 5-21, 1997, Klüwer Academic Publishers, Boston.

J.-F. Tilman, S. Truong and J.-L. Teraillon, *Optimized Distribution of Real-Time Tasks with Resource Constraints* European Space TEchnology and research Center (ESTEC), research contract report number 15235/01/NL/SVH Noordwjik, Netherlands.

M. Carlsson, G. Ottosson and B. Carlson, *An Open-Ended Finite Domain Constraint Solver3*, In Proc. of Programming Languages: Implementation, Logics, and Programs, 1997.

M. Gondran and M. Minoux, *Graphes et algorithmes*, ed. Eyrolles, 1995.

P. Van Hentenryck, V. Saraswat, and Y. Deville, *Design, Implementation and Evaluation of the Constraint Language CC(FD)*, In Constraint Programming: Basics and Trends, A. Podelski Ed., Chatillon-sur-Seine, Springer-Verlag LNCCS 910, pp. 68-90, 1995.

Improved algorithm for finding (a,b)-super solutions

Emmanuel Hebrard
National ICT Australia and
University of New South Wales
Sydney, Australia.
ehebrard@cse.unsw.edu.au

Brahim Hnich
Cork Constraint Computation Centre
University College Cork, Ireland.
brahim@4c.ucc.ie

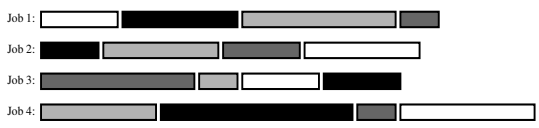
Toby Walsh
National ICT Australia and
University of New South Wales
Sydney, Australia.
tw@cse.unsw.edu.au

Abstract

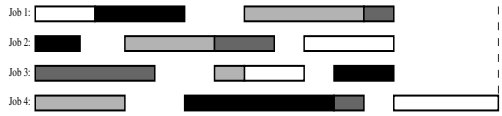
In (EHW04b), the authors introduced to constraint programming the notion of (a, b)-super solutions. They are solutions in which, if a small number of variables lose their values, we are guaranteed to be able to repair the solution with only a few changes. This concept is useful for scheduling in dynamic and uncertain environments when the robustness of the schedule is a valuable property. We introduce a new algorithm for finding super solutions that improves upon the method introduced in (EHW04a) in several dimensions. This algorithm is more space efficient as it only requires to double the size of the original constraint satisfaction problem. The new algorithm also permits us to use any constraint toolkit to solve the master problem as well as the subproblems generated during search. We also take advantage of multi-directionality and of inference based on the *neighborhood* notion to make the search for a solution faster. Moreover, this algorithm allows the user to easily specify extra constraints on the repairs.

Introduction

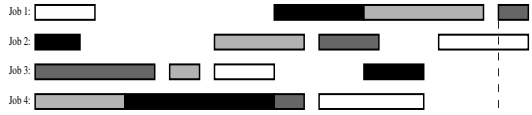
In (EHW04b), the authors introduced to constraint programming the notion of (a, b)-super solutions. Super solutions are a generalization of both fault tolerant solutions (WB98) and super models (MGR98). These are solutions such that a small (bounded) perturbation on the input will have proportionally small repercussions on the outcome. For instance when solving a scheduling problem, we may want that, in the event of a machine breaking, or of a task executing longer than expected, the rest of the schedule should change as little as possible if at all. As a concrete example, consider the following job-shop scheduling problem, where we need to schedule four jobs consisting of four activities, each requiring a different machine. The usage of a machine is exclusive, and the sequence of a job is to be respected.



The second figure shows an optimal solution.



One may argue that this solution is not robust. Indeed activities are tightly grouped and a "break" on a machine, and the subsequent delay, may trigger further delays in the schedule. On the other hand the next figure shows a solution where, for a small makespan increase, a large proportion of activities can be delayed of two units of time, without affecting at all the rest of the schedule.



Such solutions are thus *stable* in the sense that when invalidated, a close alternative solution can be applied. However, there are a number of other ways to capture robustness. for instance in (HF93), the approach to robustness is probabilistic and a robust solution is simply one that is likely to remain valid after a contingent change. The problem of scheduling under uncertainty has been widely studied in the past (See (AD01) and (NPO04) for instance). We wish to investigate how super solutions compare to these specialized methods. For instance, if we consider the *slack-based* framework (AD01), the intuitive idea is that a local perturbation will be "absorbed" if enough slack is available, end therefore solutions involving slack are preferred. Now, one can think of a scenario where the best reaction to a delay or a break would not be to delay the corresponding (plus perhaps few other) activity, but to postpone it and advance the starting time of another activity instead. This situation is not captured by slack based method. Although, it is important to notice that if the latter approach aims at minimizing the impact of a delay on the makespan, it is only a secondary consequence for super solutions. Indeed, the main concern is to limit the number of activities to reschedule. Therefore, a direct comparison is difficult. The concept of *flexible schedule* is more closely related to our view of robustness as this method promotes stability against localized perturbations.

On the other hand, super solutions have a priori several drawbacks compared to such specialized approaches.

Firstly, finding super solutions is a difficult problem and as a result the methods proposed so far are often restricted to toy problems like the one used in the previous example.

Algorithms with better performance have been proposed for very restricted classes of super solution. However, if we do not restrict ourselves to these classes, solving a problem of reasonable size is often out of reach with the current approaches.

Another difficulty is that, being a general framework, it is not always immediately applicable to a specific problem. In (EHW04a) we showed how super solutions have to be adapted to cope with specifics of job shop scheduling problems in particular. For instance, if variables are activities and values are time points, then we cannot schedule to an earlier time point as a response to a break. Moreover, moving the same activity to the next consecutive time point may not be a valid alternative.

In this paper we introduce a new algorithm that can help to address the above drawbacks. The central feature of the new algorithm is that it is closer to a "regular" solver, we solve both the original problem as well as sub-problems generated during search using any standard constraint solver. Therefore, methods that have been proven to be effective in a particular area (like shaving (CP94), specialized variable orderings (SF96) or specialized constraint propagators) can be used both for the main problem and the sub-problems. We also propose a *more effcient* and *more effective* algorithm than what has been proposed in (EHW04a). The new algorithm is more efficient as we avoid solving *all* sub-problems and it is *more effective* by using the information gathered when solving these sub-problems to get more pruning on future variables. Moreover, this architecture also helps to adapt the criteria of robustness to the problem. Indeed, to model a particular requirement we can just add it as a constraint to the sub-problems.

Formal background and notations

A constraint satisfaction problem (CSP) P consists of a set of variables $\mathcal{X}$, a set of domains $\mathcal{D}$ such that $\mathcal{D}(X_i)$ is the finite set of values that can be taken by the variable X_i, and a set of constraints $\mathcal{C}$ that specify allowed combinations of values for subsets of variables. We use upper case for variables (X_i) and lower case for values (v). A full or partial instantiation $S = \{\langle X_1 : v_1\rangle, \ldots \langle X_n : v_n\rangle\}$ is a set of assignments $\langle X_i : v_j\rangle$ such that $v_j \in X_i$. We will use $S[i]$ to denote the value assigned to X_i in S. A (partial) solution is an instantiation satisfying the constraints. Given a constraint C_V on a set of variables V, a *support* for $X_i = v_j$ on C is a partial solution involving the variables in V and containing $X_i = v_j$. A variable X_i is *generalized arc consistent* (*GAC*) on C iff every value in $\mathcal{D}(X_i)$ has support on C. A constraint C is GAC iff each constrained variable is GAC on C, and a problem is GAC iff all constraints in $\mathcal{C}$ are GAC. Given a CSP P and a subset $A = \{X_{i_1}, \ldots X_{i_k}\}$ of $\mathcal{X}$, a solution S of the restriction of P to A (denoted $P|_A$) is a partial solution on A such that if we restrict $\mathcal{D}(X_i)$ to $\{S[i]\}$ for $i \in [i_1..i_k]$, then P can be made GAC without domain wipe-out.

We introduce some notations used later in the paper. the function $H(S, R)$ is defined to be the Hamming distance between two solutions R and S, i.e., the number of variables assigned to different values in S and R. We also define $H_A(S, R)$ to be the Hamming distance restricted to the variables in A.

$$H_A(S, R) = \sum_{X_i \in A} (S[i] \neq R[i])$$

An *a-break* on a (partial) solution S is a combination of a variables among the variables in S. A *b-repair* of S is a (partial) solution R such that $H_A(S, R) = |A|$ and $H(S, R) \leq (a + b)$. In other words, R is an alternative solution for S such that if the assignments of the variables in A are forbidden, the remaining "perturbation" is restricted to b variables.

Definition 1 *A solution S is an (a, b)-super solution iff for every $a' \leq a$, and for every a'-break of S, there exists a b-repair of S.*

The basic algorithm

We first describe a very simple and basic version of the algorithm without any unnecessary features. Then we progressively introduce modifications to make the algorithm more efficient and more effective.

The basic idea is to ensure that the current partial solution is also a partial super solution. In order to do so, as many sub-problems as possible breaks for this partial solution have to be solved. The solutions to these sub-problems are partial repair solutions. We therefore work on a copy of the original problem that we change and solve for each test of *reparability*. Note that the sub-problem is much easier to solve than the main problem, for several reasons. The first reason is that each of the sub-problems is polynomial. Indeed, since a repair solution must have less than $a + b$ discrepancies with the main solution, the number of possibilities is bounded by $n^{a+b}d^{a+b}$. Typically, the cost of solving one such sub-problem will be far below this bound since constraint propagation is used. Another reason is that we can reuse the variable ordering dynamically computed whilst solving the main problem. Furthermore, we will demonstrate later that not all breaks have to be checked, and that we can infer inconsistent values from the process of looking for a repair. Pruning the main problem is critical, as it not only reduces the search tree, but also reduces the number of sub-problems to be solved.

The algorithm for finding super solutions is, in many respects, comparable to a global constraint. However, it is important to notice that we *cannot* define a global constraint ensuring that the solution returned is a super solution as this condition depends on the variables domains, whilst constraints should only depend on the values assigned to variables and not their domains. For example, consider two variables X_1, X_2 taking values in $\{1, 2, 3\}$ such that $X_1 < X_2$. The assignment $\langle X_1 = 1, X_2 = 3\rangle$ is a $(1, 0)$-super solution, however, if the original domain of X_1 is $\{1, 3\}$, then $\langle X_1 = 1, X_2 = 3\rangle$, is *not* a $(1, 0)$-super solution. Nevertheless, the algorithm we introduce could be seen as a global constraint implementation as it is essentially an oracle tightening the original problem It is however important to ensure that this oracle is not called for every change in a domain as it can be costly.

Initialization: In Algorithm 1, we propose an pseudo code for finding super solutions. The input is a CSP, i.e., a triplet $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and the output a (a, b)-super solution S. We first create a copy P' of P, where $X'_i \in \mathcal{X}'$ iff $X_i \in \mathcal{X}$ and $C' \in \mathcal{C}'$ iff $C \in \mathcal{C}$. This copy will be used to find b-repairs. At any point in the algorithm, $\mathcal{D}(X_i)$ (resp. $\mathcal{D}'(X'_i)$) is the *current* domain of X_i (resp. X'_i). The set $Past \subseteq \mathcal{X}$ contains all variables that are already bound to a value and we denote $Past'$ the set containing the same variables, but "primed", $Past' = \{X'_i | X_i \in Past\}$.

Algorithm 1: `super-MAC`(P, a, b)

```
Data   : P, a, b
Result : S: an (a, b)-super solution
S ← ∅; Past ← ∅; P' ← P;
if ¬backtrack(P, P', S, Past, a, b) then
    print "NO SUPER-SOLUTION";
print "A SUPER-SOLUTION FOUND: S";
```

Main Backtracker Procedure (Algorithm 2) searches and backtracks on the main problem P. It is in very similar to a classical backtracker that maintain GAC at each node of the search tree, except that we also add a call to the procedure `reparability` at each node. Note that any solver or local/global consistency property can be used instead as long as the procedure `reparability` is called. A possible way of implementing `reparability` –in a standard constraint toolkit– can be as a global constraint containing internally the extra data structure P' and an associated specialised solver. As such global constraint can be costly, it should not be called more than once per node.

Algorithm 2: `backtrack`$(P, P', S, Past, a, b)$: Bool

```
if Past = X then return True;
choose X_i ∈ X \ Past;
Past ← Past ∪ {X_i};
foreach v ∈ D(X_i) do
    save D;
    D(X_i) ← {v};
    S ← S ∪ {⟨X_i : v⟩};
    if AC-propagate(P) & reparability(P, P', S, Past, a, b)
    then
        if backtrack(P, S, Past, a, b) then return True;
    restore D;
    S ← S − {⟨X_i : v⟩};
Past = Past − {X_i};
return False;
```

Enforcing reparability: Procedure `reparability` (Algorithm 3) makes sure that each a-break of the solution S has a b-repair. If $|S| = k$ then we check all combinations of less than a variables in S, that is $\sum_{j \leq a} \binom{k}{j}$ breaks. This number has no closed form, though it is bounded above by k^a. For each a-break, we model the problem of the existence of a b-repair using P'. Given the main solution S and a break A, we need to find a b-repair, that is, a solution R of $P'|_{Past'}$ such that $H_A(S, R) = |A|$ and $H(S, R) \leq |A| + b$. The domains of all variables are set to their original state. Then for any $X'_i \in A$, we remove the value $S[i]$ from $\mathcal{D}'(X'_i)$, thus making sure that $H_A(S, R) = |A|$. We also add an AtMost$k$Diff constraint that ensures $H(S, R) \leq k$, where $k = |A| + b$. Finally, we solve $P'|_{Past'}$, it is easy to see that any solution is a b-repair.

Algorithm 3: `repairability`$(P, S, Past, a, b)$:Bool

```
foreach A ⊆ Past' such that |A| ≤ a do
    foreach X_i ∈ A do
        D'(X'_i) ← D(X_i) − {S[i]};
    k ← (|A| + b);
    S' ← solve(P'|_Past' + AtMostkDiff(X', S));
    if S' = nil then return False;
return True;
```

Propagating the AtMost$k$Diff constraint: The AtMost$k$Diff constraint is defined as follows:

Definition 2 AtMost$k$Diff$(X'_1, \ldots X'_n, S)$ *holds iff* $k \geq \sum_{i \in [1..n]} (X'_i \neq S[i])$

This constraint ensures that the solution we find is a valid partial b-repair by constraining the number of discrepancies to the main solution to be lower than $a + b$. To enforce GAC on such a constraint, we first compute the smallest expected number of discrepancies to S. Since S is a partial solution we consider the *possible* extensions of S. Therefore, when applied to the auxiliary CSP P' this number is simply

$$d = |\{i | \mathcal{D}'(X'_i) \cap \mathcal{D}(X_i) = \emptyset\}|$$

We have three cases:

1. If $d < k$ then the constraint is GAC as every variable can be assigned any value providing that all other variables X'_i take a value included in $\mathcal{D}(X_i)$, and we will still have $d \leq k$.
2. If $d > k$ then the constraint cannot be satisfied.
3. If $d = k$ then we can set the domain of any variable X'_i such that $\mathcal{D}(X'_i) \cap \mathcal{D}(X_i) \neq \emptyset$ to $S[i]$.

Comparison with previous algorithm This new algorithm is simpler than the one given in (EHW04a) as no extra data structure is required for keeping the current state of a repair. Moreover, the space required is at most twice the space required for solving the original problem, whilst the previous algorithm stored the state of each search for a b-repair. We want to avoid such data structures as they are exponential in a. Even though a is typically a small constant, this can be prohibitive. Another advantage in doing so is that the search for a repair can easily be done, whereas in the previous algorithm, doing so would have been difficult without keeping as many solver states as breaks, since the search was starting from the point it ended in the previous call.

The search tree explored by this algorithm is strictly smaller than that explored by the previous algorithm. The

methods are very comparable as they both solve sub-problems to find a b-repair for each break. However, since the sub-problems solved by this previous algorithm were implemented as a simple backtrack procedure (without constraint propagation), it was not possible to check if a b-repair would induce an arc inconsistency in an unassigned variable.

Improvements

Now we explore several ways of improving the basic algorithm.

Repair multi-directionality

The multi-directionality is a concept used for instance for implementing general purpose algorithms for enforcing GAC. The idea is that a tuple is a support for every value it involves. The same metaphor applies when seeking repairs instead of supports. In our case, suppose that we look for a $(2, 2)$-super solution and suppose that R is a repair solution for a break on the variables $\{X, Y\}$ that require reassigning the variables $\{V, W\}$. This constitutes also a repair for $\{X, V\}, \{X, W\}, \{Y, V\}, \{Y, W\}$ and $\{V, W\}$. We therefore need not to look for repair for these breaks.

We used a simple algorithm from Knuth (Knu) to generate all $\leq a$-breaks. This algorithm generates the combinations in a deterministic manner, and therefore constitutes an ordering on these combinations. Moreover, this ordering has the nice property that given one combination in input, one can compute the rank of this combination in the ordering in linear time on the size of the tuple. The size of the tuple is in our case a small constant, we thus have an efficient way of knowing if the break that we currently consider is covered by an earlier repair. Each time a new repair is computed, all breaks it covers are added to a set, then when we generate a combination, we simply check that its index is not in this set otherwise we do not need to find a repair for it.

Neighborhood-based inference:

The second observation that we make to improve the efficiency is less obvious but has broader consequences. First, let us introduce some necessary notation:

A path linking two variables X and Y is a sequence of constraints $C_{V_1}, \ldots C_{V_k}$ such that $i = j + 1 \Rightarrow V_i \cap V_j \neq \emptyset$ and $X \in V_1$ and $Y \in V_k$, k is the length of the path. The distance between two variables $\delta(X, Y)$ is the length of the shortest path between these variables ($\delta(X, X) = 0$). $\Delta_d(X)$ denotes the neighborhood at a distance exactly d of X, i.e., $\Delta_d(X) = \{Y \mid \delta(X, Y) = d\}$. $\Gamma_d(X)$ denotes the neighborhood up to a distance d of X i.e., $\Gamma_d(X) = \{Y \mid \delta(X, Y) \leq d\}$. Similarly, we define the neighborhood $\Gamma_d(A)$ (resp. $\Delta_d(A)$) of a subset of variable A as simply $\bigcup_{X \in A} \Gamma_d(X)$ (resp. $\Delta_d(A)$).

Now we can state the following lemma which will be central to all the subsequents improvements. It shows that if there exists a b-repair for a particular a-break A, then all reassignments are within the neighborhood of A up to a distance b.

Lemma 1 *Given a solution S and a set A of a variables, the following equivalence holds:*

$$\exists R \text{ s.t. } (H(S|_A, R|_A) = a \text{ and } H(S, R) < d)$$
$$\Leftrightarrow$$
$$\exists R' \text{ s.t. } (H(S|_A, R'|_A) = a \text{ and } H(S|_{\Gamma_{d-a}(A)}, R'|_{\Gamma_{d-a}(A)}) = H(S, R') < d)$$

Proof: We prove this lemma constructively. We start from two solutions S and R that satisfy the premise of this implication and construct R' such that S, R' satisfy the conclusion. We have $H(S, R) = k_1 < k$, therefore exactly k_1 variables are assigned differently between S and R. We also know that $H(S|_A, R|_A) = |A| = a$ therefore only $b = k_1 - a$ are assigned differently outside A. Now we change R into R' in the following way. Let d be the smallest integer such that $\forall X_i \in \Delta_d(A),\ R[i] = S[i]$. It is easy to see that $d \leq b$ as $\Delta_{d_1}(A)$ and $\Delta_{d_2}(A)$ are disjoint iff $d_1 \neq d_2$. We let all variables in $\Gamma_d(A)$ unchanged, and for all other variables we set $R'[i]$ to $S[i]$. Now we show that R' satisfies all constraints. Without loss of generality, consider any constraint C_V on a set of variables V. By definition, the variables in V belongs to at most two sets $\Delta_{d_1}(A)$ and $\Delta_{d_2}(A)$ such that d_1 and d_2 are consecutive (or possibly $d_1 = d_2$). We have 3 cases:

1. $d_1 \leq d$ and $d_2 \leq d$: all variables in V are assigned as in R, therefore C_V is satisfied.
2. $d_1 > d$ and $d_2 > d$: all variables in V are assigned as in S, therefore C_V is satisfied.
3. $d_1 = d$ and $d_2 = d + 1$: the variables in $\Delta_{d_2}(A)$ are assigned as in S, and by definition of R', the variables in $\Delta_{d_1=d}(A)$ are assigned as in S, therefore C_V is satisfied.

$\diamond$

Computing this neighborhood is not time expensive, as it can be done as a preprocessing step. A simple breadth first search on the constraint graph, i.e., the graph were any two variables are connected iff they are constrained by the same constraint. The neighborhood $\Gamma_d(A)$ of a break A is recomputed each time, however it just requires a simple union operation over the neighborhood of the elements in A.

Updates of the auxiliary CSP: The first use of Lemma 1 is straightforward. We know that, for a given break A, there exists a b-repair only if there exists one that sharea all assignments outside $\Gamma_b(A)$. Therefore, we can make P' equal to the current state of P apart from $\Gamma_b(A)$. This does not make the algorithm stronger. However, we can then post an ATMOSTkDIFF constraint only on $\Gamma_b(A)$ instead of $\mathcal{X}'$, since we have all the pruning on $\mathcal{X}' \setminus \Gamma_b(A)$ for "free".

Avoiding useless repair checks: Now suppose that $\Gamma_{b+1}(A) \subseteq Past'$. Then we know that this break has already been checked at the previous level in the search tree, and the repair that we found has the property that assignments on $\Delta_{b+1}(A)$ are the same as in the current solution. Thus any extension of the current partial solution, is also a valid extension of this repair. Therefore we know that

this repair will hold in any subtree, hence we do not need to check it unless backtracking beyond this point.

Tightening the ATMOSTkDIFF **constraint:** Considering the property of Lemma 1, we can tighten the ATMOSTkDIFF constraint by forbidding some extra tuples. Doing this, we get a tighter pruning when doing arc consistency, while keeping at least one solution, if such solution exists. The first tightening is that all differences outside $\Gamma_b(A)$ are forbidden. But, we can do even more inference. For instance, suppose that for that we look for a 3-repair for the break $\{X'_1\}$ and that $\Delta_1(X'_1) = \{X'_2, X'_3\}, \Delta_2(X'_1) = \{X'_4\}, \Delta_3(X'_1) = \{X'_5, \}$, and the domains are as follows:

$$X'_1 = \{1, 2\}, X'_2, X'_3 = \{3, 4\}, X'_4, X'_5 = \{1, 2, 3, 4\}$$

Moreover, suppose that for the main backtracker, the domains are as follows:

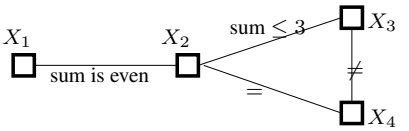
$$X_1 = 3, X_2, X_3, X_4, X_5 = \{1, 2\}$$

We can observe that already 2 reassignments are made at distance 1 from X_1. As a consequence, if X'_5 was to be assigned differently to X_5, then X_4 would have to be equal to X_4, and therefore there is no discrepancy on any variable from $\Delta_2(X'_1)$, hence there must be a repair such that any variable outside $\Gamma_2(X_1)$ is assigned as in the main solution. We can thus prune the values 3 and 4 from $\mathcal{D}(X'_5)$ (to make it equal to X_5).

Inference from repair-seeking to the main CSP: We can infer that some values of the main CSP will never participate in a super solution while seeking for repairs. This allows us to prune the future variables, which can greatly speed up the search process, especially when combined with GAC propagation on "regular" constraints. For instance consider constraint problem P, composed of the domain variables:

$$X_1 = \{1, 2, 4\}, X_2 = \{1, 2\}, X_3 = \{1, 2\}, X_4 = \{1, 2\}$$

subject to the following constraint network:



We have $P' = P$, and it is easy to see that P is arc consistent. Now suppose that we look for a $(1, 1)$-super solution, and our first decision is to assign the value 1 to X_1. The domains are reduced so that P remains arc consistent:

$$X_1 = \{1\}, X_2 = \{1\}, X_3 = \{1, 2\}, X_4 = \{1, 2\}$$

Then we want to make sure that there exist a 1-repair for the break $\{X_1\}$. We then consider P' where $\mathcal{D}(X'_1)$ is set to $\mathcal{D}(X'_1) \setminus \{1\}$. Moreover the constraint ATMOST2DIFF is posted on $\Gamma_1(\{X_1\}) = (X_1, X_2)$:

$$X'_1 = \{2, 4\}, X'_2 = \{1, 2\}, X'_3 = \{1, 2\}, X'_4 = \{1, 2\}$$

Since $P'|_{\{X'_1\}}$ is satisfiable, (for instance, $\{\langle X'_1 : 2\rangle\}$ is a partial solution that does not produce a domain wipe out in any variable of P') we continue searching. However, if before solving P' in order to find a repair we first propagate arc consistency, then we obtain the following domains:

$$X'_1 = \{2, 4\}, X'_2 = \{2\}, X'_3 = \{1\}, X'_4 = \{2\}$$

Observe that $2 \in \mathcal{D}(X_3)$ whilst $2 \notin \mathcal{D}(X'_3)$, this means that no repair for X_1 can assign the value 2 to X_3. However, Lemma 1 works in both direction, since $X'_3 \notin \Gamma_1(X'_1)$, we can conclude that X'_3 and X_3 should be equal, and therefore we can prune the value 2 directly from X_3. In this toy example, this removal will make P arc inconsistent, and therefore we can conclude without searching that X_1 cannot be assigned to 1.

Notice that this extra pruning comes at no extra cost, the only condition that we impose is to make P' arc consistent *before* searching on it. After this arc consistent preprocessing for a break A, any value pruned from the domain of a variable $X'_i \in \mathcal{X}' \setminus \Gamma_b(A)$, can be pruned from X_i as well.

The main drawback of this method is that as soon as the problem involves a global constraint (for instance "all the variables must be different"), then typically we have $\Gamma_1(X_i) = \mathcal{X}$ for any $X_i \in \mathcal{X}$. Therefore all previous improvements based on neighborhood are useless. However, one can make such inference, but using a different reasoning, even in the presence of large arity constraints. The idea is the following: Suppose that after enforcing GAC on P', the least number of discrepancies is exactly $a + b$, that is, $Diff = \{i | \mathcal{D}(X_i) \neq \mathcal{D}'(X'_i)\}$ & $|Diff| = a + b$. We can deduce that any variable X'_j such that $j \notin Diff$ must be equal to X_j, for *any* b-repair. Indeed, it applies to any repair, since only pre-processing and no search was used. Therefore, we can prune domains in both P and P' as follows:

$$\forall i \notin Diff\ \mathcal{D}(X_i) \leftarrow \mathcal{D}(X_i) \cap \mathcal{D}(X'_i)\ \&\ \mathcal{D}(X'_i) \leftarrow \mathcal{D}(X_i)$$

We can therefore modify `reparability` by taking into account the previous observations (Algorithm 4).

Extensions

In (EHW04a), the authors propose to extend or restrict super solutions in several directions to make them more useful practically. In scheduling problems, we may have restrictions on how the machines are likely to break, or how we may repair them. Furthermore, we have an implicit temporal constraint that forbid some reassignments. For instance, when a variable breaks, there are often restrictions on the alternative value that it can take. When the values represent time, then an alternative value might have to be larger than the broken value. Alternatively, or in addition, we may want the repair to be chosen among larger values, for some ordering. It may also be the case that certain values may not be brittle and so cannot break. Or that if certain values are chosen, they cannot be changed. This algorithm allows to express these restrictions (and many more) very easily, as they can be modelled as extra constraints on P'.

For instance to model the fact that an alternative value has to be larger than the broken value, one can post the unary

Algorithm 4: `repairability`$(P, S, Past, a, b)$:Bool

```
covered ← ∅;
foreach A ⊆ Past' such that |A| = a and A ∉ covered and Γ_{b+1} ⊈ Past' do
    foreach X_i ∈ A do
        D'(X'_i) ← D'(X'_i) − {S[i]};
    foreach X_i ∈ (X \ Γ_b(A)) do
        D'(X'_i) ← D(X_i);
    if ¬AC-propagate(P') then return False;
    Diff ← {i | D(X_i) ≠ D'(X'_i)};
    if Diff = a + b then
        foreach X_i ∈ (X \ Diff) do
            D(X_i) ← D'(X'_i) ← (D'(X'_i) ∩ D(X_i));
    foreach X_i ∈ (X \ Γ_b(A)) do
        D(X_i) ← D'(X'_i);
    S' ← solve(P'|_{Past'} + AtMost(|A| + b)Diff(Past', S));
    if S' = nil then return False;
    Diff ← {i | S'[i] ≠ S[i]};
    foreach A' ⊆ Diff such that |A'| = a do
        covered ← covered ∪ {A};
```

constraint $X > S[X]$, where X is any variable involved in the break, and $S[X]$ is the value assigned to this variable in the main solution. Moreover, one can change the constraint AtMost$k$Diff itself. For instance, suppose that we are only interested in the robustness of the overall makespan, and we are solving a sequence of *deadline job shop*. One can extend the deadline of P' by a given (acceptable) quantity q, and omit the AtMost$k$Diff constraint. The resulting solution will be one such that no "break" of size less than or equal to a can result in a makespan increase of more than q.

Another concern is that the semantic of the super solution depends on the model chosen for solving a problem. For instance, an efficient way of solving the job shop scheduling problem is to search over sequences of activities on each resource, rather than assigning start times to activities. In this case, two solutions involving different start times may map to a single sequence. Therefore the semantic of a super solution is changed, a break or a repair implies a modification of the order of activities for a resource. It is therefore interesting to think of ways of solving a problem using a model whilst ensuring that the solution is a super solution for another model. If it is possible to channel both representations, then one can solve one model whilst applying the `reparability` procedure to the second model.

Optimization

Finding super solutions is still, and will certainly remain a difficult problem. One way to avoid this difficulty is to try to get a solution as close as possible to a super solution. We can then start from an initial solution found using the best available method, and then we improve its *reparability* with a branch and bound algorithm.

The (a, b)-*reparability* of a solution is defined as the number of combinations of less than a variables that are covered by a b-repair. In (EHW04a), the authors report that turning the procedure into a branch and bound algorithm that maximize reparability is the most promising way of using super solutions in practice as an (a, b)-super solution may not always exist. Moreover doing so, one can get a "regular" solution with the fastest available method, and improve its reparability afterward.

The algorithm introduced here can easily be adapted in this way. The procedure `reparability` would return the number of b-repairs founds, instead of failing when a break does not accept one. The main backtracker would then backtrack either if the problem is made arc inconsistent or the value returned by `reparability` is less than the reparability of the best full solution found so far. We rewrite the procedure `reparability` adapted to this purpose in Algorithm 5.

Algorithm 5: `reparability`$(P, S, Past, a, b)$:Int

```
covered ← ∅;
foreach A ⊆ Past' such that |A| = a and A ∉ covered and Γ_{b+1} ⊈ Past' do
    foreach X_i ∈ A do
        D'(X'_i) ← D'(X'_i) − {S[i]};
    foreach X_i ∈ (X \ Γ_b(A)) do
        D'(X'_i) ← D(X_i);
    S' ← solve(P'|_{Past'} + AtMost(|A| + b)Diff(Past', S));
    if S' ≠ nil then
        Diff ← {i | S'[i] ≠ S[i]};
        foreach A' ⊆ Diff such that |A'| = a do
            covered ← covered ∪ {A};
return |covered|;
```

Unfortunately, if all other improvements still hold, we cannot prune P as a result of a pruning when pre-processing P', since a break without repair is allowed.

Future work

Our next priority is to implement this algorithm for finding super solutions to the job-shop scheduling problem. We will use a constraint solver that implements shaving, the Orr heuristic described in (SF96), as well as constant time propagators for enforcing GAC on precedence and overlapping constraints. As these two constraints are binary, the neighborhood of a variable is limited. Hence the theoretical results introduced here should apply. Moreover, we expect this reasoning to offer a good synergy with a strong global consistency method such as shaving. Indeed more inference can be done on the repairs (without searching) than with GAC, and thus more values can be pruned in the main search tree because of the robustness requirement. We therefore expect to be able to solve much larger problems than the instances solved in (EHW04a).

Conclusion

We have introduced a new algorithm for finding super solutions that improves upon the previous method. The new algorithm is more space efficient as it only requires to double the size of the original constraint satisfaction problem. The new algorithm also permits us to use any constraint toolkit to

solve the master problem as well as the sub-problems generated during search. We also take advantage of repair multi-directionality and of inference based on just a restricted *neighborhood* of constraints. Moreover, this algorithm allows the user to easily specify extra constraints on the repairs. For example, we can easily specify that all repairs should be later in time than the first break.

Acknowledgements

Emmanuel Hebrard and Toby Walsh are supported by National ICT Australia. Brahim Hnich is currently supported by Science Foundation Ireland under Grant No. 00/PI.1/C075. We also thank ILOG for a software grant.

References

J.C. Beck A.J. Davenport, C. Gefflot. Slack-based techniques for robust schedules. In *Proceedings ECP'01*, 2001.

J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European journal of Operational Research*, 78:146–161, 1994.

B. Hnich E. Hebrard and T. Walsh. Robust solutions for constraint satisfaction and optimization. In *Proceedings ECAI'04*, 2004.

B. Hnich E. Hebrard and T. Walsh. Super solutions in constraint programming. In *Proceedings CP-AI-OR'04*, 2004.

J. Lang H. Fargier. Uncertainty in constraint satisfaction problems: a probabilistic approach. In *Proceedings EC-SQARU'93*, 1993.

D. Knuth. The art of computer programming: Prefascicle 3a: Generating all combinations. http://www-cs-faculty.stanford.edu/ knuth/fasc3a.ps.gz.

A. Parkes M. Ginsberg and A. Roy. Supermodels and robustness. In *Proceedings AAAI-98*, pages 334–339, 1998.

A. Cesta N. Policella, S.F. Smith and A. Oddi. Generating robust schedules through temporal flexibility. In *Proceedings ICAPS'04*, 2004.

N. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86(1):1–41, September 1996.

R. Weigel and C. Bliek. On reformulation of constraint satisfaction problems. In *Proceedings ECAI-98*, pages 254–258, 1998.

Constraint-Based Envelopes over Multiple Alternatives

Tatiana Kichkaylo*
USC/Information Sciences Institute
Marina del Rey, CA 90292, USA
tatiana@isi.edu

Abstract

Many real-world planning problems require sophisticated reasoning about numeric resources, including sharing of resources by different actions. Constraint-based planners offer machinery to represent complex constraints and dependencies between actions. However, such planners usually consider a single partial plan at a time, and limit themselves with finding a feasible solution. Alternatively, planning graph based planners use an aggregate data structure over multiple partial plans to guide search, which allows such planners to guarantee optimality of the solution. Typically, such planners restrict the form of supported resource expressions. In this paper we describe a planner for construction of computational grid workflows that combines these approaches. Our planner uses an envelope over multiple execution sequences represented by a constraint network to drive the search towards a good solution, while supporting expressive models for resource sharing and multi-resource reservations.

Introduction

Traditionally, in planning, resources are allocated to actions using an all-or-none policy; they cannot be shared. For example, at most one job can be executed on a machine at any given time.

In real-world planning problems, the amount of available resources and action requirements are often stated using real numbers. This more general model of resources creates additional degree of flexibility in the problem. For example, the number of jobs that can be executed on a machine in parallel depends on the resource requirements of the jobs and the amount of resources available on the machine. The situation becomes even more complicated when actions have durations and the resource availability changes over time.

In planning, the sequence of actions to be scheduled first needs to be chosen. Often, the same result may be achieved in different ways. The resource requirements of different options for achieving intermediate goals can interact in complex ways, affecting feasibility and optimality of the solution. For example, a job may require a host with at least 16 CPUs. The execution time of such a job depends on the number of CPUs allocated for the job and their speed. Allocation of a larger number of CPUs to one job may decrease its running time, but make parallel execution of other jobs impossible.

Constraint-based planners (Smith, Frank, & Jónsson 2000; Rabideau, Engelhardt, & Chien 2000; Ghallab & Laruelle 1994) support very expressive resource models. However, such planners usually refine a single partial plan at any given moment and focus on finding a feasible solution. To produce solutions of good quality (minimum makespan, smaller resource consumption), such planners rely on domain-specific control knowledge, which needs to be provided by a human expert.

On the other hand, GraphPlan-based planners (Blum & Furst 1997; Koehler 1998; Gerevini & Serina 2002) build a data structure, called a planning graph, that aggregates information over multiple partial plans, essentially providing a logical envelope over all possibly reachable world states. Planning graphs help to find an optimal solution. Unfortunately, GraphPlan-based planners offer limited support for complex resource interactions and time-varying resource availability.

In this paper we describe GPRS (Grid Planner with Reservations and Sharing), a planner for constructing executable workflows in computational grid environments. Our planner combines ideas from planning graph and constraint-based planners. GPRS uses an envelope over multiple alternative partial plans for search guidance. To represent sharing and reservation of resources, numeric variables belonging to nodes of the envelope graph are organized in a constraint network. This allows GPRS to guide search towards an optimal solution while supporting expressive models for resource sharing and multi-resource reservations.

The rest of this paper is structured as follows. First, we present the problem of constructing grid workflows. Then, we describe the GPRS algorithm, and evaluate (i) its ability to handle expressive resource models, (ii) scalability of the planner, and (iii) quality of the solutions. We describe related work, and conclude with a discussion of limitations of our algorithm and future research directions.

*The work was done when the author was a student at New York University.

Workflow construction problem

The workflow construction problem (WCP) is the problem of designing executable workflows in computational grids. The objective of computational grids is to pool together distributed computational and storage equipment to efficiently solve computationally and data intensive tasks.

A typical grid environment consists of a network of hosts and links, which have resources, such as CPU, memory, and bandwidth. We model resource availability as a piecewise constant function of time.

A typical grid workflow consists of jobs that process files. A file has a globally unique name and a size. Each job is specified by a set of required files, a set of produced files, and resource requirements. A job can be executed on a host that has sufficient resources. The resources are occupied for the duration of the job and released upon its completion.[1] In general, the duration of the job is a function of the amount of resource available for the job.

Files can be transfered over network links, consuming link resources for the duration of the transfer. The latter depends on the size of the file and on the available network bandwidth. Note that it is possible to transfer files over multi-link paths. In this case, the duration is computed using the minimum link bandwidth along the path (path bandwidth). During the transfer, path bandwidth is reserved on all participating links, which allows high-bandwidth links to be shared between several parallel low-bandwidth transfers.

The workflow construction problem is, given a network topology and resource availability, a set of job descriptions, and an initial allocation of files on hosts, construct an executable workflow (execution plan) consisting of job executions on hosts and file transfers over paths such that a given file is obtained on a given network host as soon as possible. Constructing an optimal executable workflow may require resource sharing and trading off computation and communication.

The WCP can be viewed as a planning problem, where job executions and file transfers correspond to actions, and resource and file availability describe the state of the world.

In the example shown in Figure 1, the goal is to obtain a small file `Res` on host 3. `Res` can be computed in 1 time unit given two files `pA` and `pB` (for the purpose of this example we assume that the job duration is the same for all hosts). Each of these files can be computed in 2 time units from files `rA` and `rB` respectively. All four files, `pA`, `pB`, `rA`, and `rB`, are present in the network. Each of these files has size 100 units. The size of the `Res` file is 10 units.

Given the available link bandwidth, the optimal strategy (Figure 2) is to transfer `pA` from the remote host over path 1-2-3-4 (action **trpA**), recompute `pB` using the `rB` file (action **jobB**), compute `Res` on host 4 (**jobC**), and transfer the final result to the destination host (**trRes**). Note that, if the execution of the component producing the final result is co-located with the recomputation of `pB`, the time used to transfer `pA` overlaps with the computation of `pB`.

Finding this optimal solution requires reasoning about resource reservations and sharing, as well as an ability to compare different possible solutions.

[1] Some resources, such as quotas for computing time in supercomputer centers, are consumed, rather than occupied and released. Currently, GPRS does not deal with these.

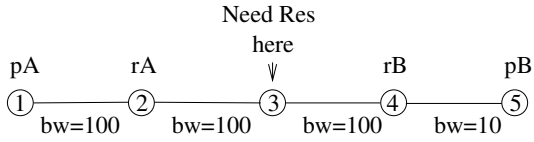


Figure 1: Bandwidth (bw) and file availability.

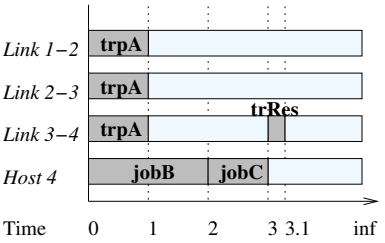


Figure 2: Solution to the problem shown on Figure 1 that trades of computation and communication.

GPRS

The Grid Planner with Reservations and Sharing (GPRS) combines the ideas of planning graph based algorithms and constraint propagation. GPRS builds an envelope over possible execution sequences to obtain completion time estimates for search guidance. Nodes of the envelope graph, which correspond to actions and propositions, may have multiple variables associated with them. The values of these variables are propagated using a constraint network. Currently, the plan extraction phase of GPRS is based on critical path scheduling.

Envelope graph

The purpose of the envelope graph is to obtain lower bound estimates on the completion time of various parts of the computation to guide the search towards the best (fastest) ways of achieving goals.

The graph has two types of nodes: **AND nodes** correspond to actions (job executions, file transfers), and **OR nodes** to propositions. In planning for computational grids, propositions describe the availability of a file on a network host.

A typical goal of a grid application is to produce a particular file on a particular host. While the size of such a goal is usually small (e.g., one file-host pair), the total amount of information about the state of the network and resource availability may be large. Therefore, GPRS constructs the envelope graph using regres-

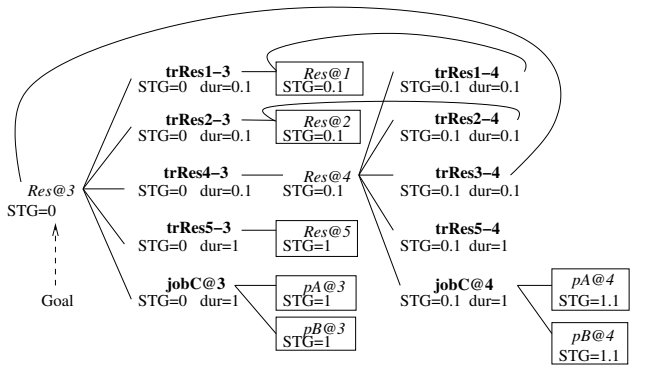


Figure 3: Part of the logical envelope graph. Bold font shows actions, italics shows propositions. Boxes correspond to leaf nodes. Numbers under nodes show shortest time to goal (STG) and action duration.

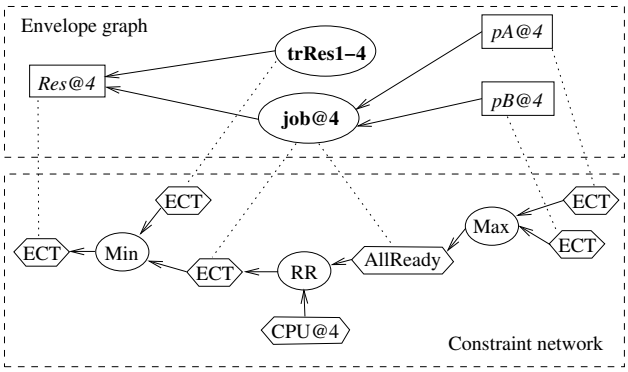


Figure 4: Envelope and constraint network. Dotted lines connect variables (hexagons) to the nodes they belong to.

sion from the goals, which allows the planner to identify and bring in necessary information during the search. Figure 3 shows the initial phase of construction of the envelope graph for the problem shown in Figure 1.

For each OR node n_o, the **support** of n_o is a set of AND nodes corresponding to actions that can achieve the proposition of the node n_o. For example, availability of file `Res` on host 3 *Res@3* is supported by four actions for transferring the file from different network hosts and one job execution action. For each AND node n_a, the support of n_a is a set of OR nodes corresponding to preconditions of the operator of node n_a. Thus, action **jobC@3** is supported by availability of files `pA` and `pB` on host 3 (*pA@3* and *pB@3*).

Both AND and OR nodes can support multiple sinks. For example, *Res@2* supports **trRes2-3** and **trRes2-4**. Such reuse of nodes makes the envelope a general graph (with cycles) and reduces memory requirements for envelope construction.

Each of the nodes of the envelope graph may have several variables associated with it, including the *Earliest Completion Time* (ECT) and the *Shortest Time to Goal* (STG) variables. ECT corresponds to the lower bound on time for completing the current node after the start of the workflow. STG is the lower bound on time to reach the goal after completion of the current node. The planner optimizes the makespan of the computation and uses ECT variables to guide the search. The STG variables are used to order constraints for propagation as described below.

A **support tree** is defined recursively as a single support node for each proposition (OR) node and all support nodes for an action (AND) node.[2] The **best tree** of the envelope graph is a support tree in which the cheapest support node is chosen for every proposition. The envelope graph is expanded until the best tree is completely rooted in the initial state.

[2]Strictly speaking, a support tree is not necessarily a tree, but a DAG.

Only leaf nodes belonging to the best tree are expanded. After expansion of a leaf node in the regression way, constraint propagation is performed, which can improve the lower bound estimates of the ECTs of nodes and change the set of nodes forming the best tree. Because of this non-uniform expansion of the envelope graph, GPRS essentially performs best-first search in the space of support trees (as opposed to GraphPlan (Blum & Furst 1997), which extends all branches of the envelope simultaneously).

Note that the envelope and the corresponding constraint network are optimistic in that they can miss dependencies between different branches. For example, logically independent jobs may compete for the same resources and, therefore might have to be scheduled sequentially rather than in parallel. Because of this optimism, the best tree of the envelope graph does not necessarily correspond to a valid solution, and additional expansion may be initiated during solution extraction.

Constraint propagation

Nodes of the constraint network, underlying the envelope graph of GPRS, may correspond to variables of the envelope nodes, such as earliest availability time of files on hosts (ECT for files), resource variables, and artificial variables.

Variables and constraints. For the ECT variables, the value of a variable is a number describing the current lower bound estimate of the completion time. Values of the resource variables may be represented as piecewise constant functions describing the maximum available levels of the resource as a function of time. Ultimately, the type of values variables can have is limited only by the constraint-propagation algorithm used.

Actions of a plan can be executed sequentially or in parallel. In both cases, the dependencies between the actions are represented using constraints.

For example, an execution of a job on a host can only start when all required files are available on that host.

This dependency is represented as follows (Figure 4). An artificial variable *AllReady* is created, which is connected to the ECT variables of all nodes corresponding to the required files using a Max constraint. The ECT variable of the job node is connected to the *AllReady* variable using a Resource Reservation (RR) constraint, which also takes into account resource availability. The reservation constraint directly computes the earliest completion time of the job given resource availability profiles, file availability time (*AllReady*), and an expression describing running time of the job as a function of available resources. The default implementation of the RR constraint searches for the earliest moment after *AllReady* when all required resources simultaneously satisfy job requirements for the duration of the job. Alternative implementations and types of constraints can be provided by the user to encode resource reservation policies specific for job types and capabilities of host operating systems/schedulers.

Actions that can be executed in parallel may affect each other via shared resources. The envelope of GPRS is built over multiple possible execution sequences, and not all actions that are included in the envelope, or even in its current best tree, would necessarily be incorporated in the final plan. Therefore, GPRS takes an optimistic approach. Only reservations of nodes chosen during solution extraction to be a part of the solution (committed nodes) affect resource availability visible to other actions. In other words, a resource can be promised to several actions at the same time. This can cause suboptimality of a solution.

Scheduling of constraint propagation. Constraint propagation is performed after every change to the envelope graph: expansion of a leaf node or commitment of a node during solution extraction. Since constraint propagation is the most frequent and expansive operation, it is desirable to make it as fast as possible.

GPRS requires that every variable belongs to at most one envelope node (resource variables are independent), and every constraint changes values of variables of a single node. The latter restriction can be enforced by creating artificial variables. The nodes of the envelope graph, in addition to ECT variables, also own Shortest Time to Goal (STG) variables (Figure 3), which are lower bounds on the completion time of the plan after completion of execution of the node. The values of STG variables are obtained by simple summation of lower bounds of action durations during the expansion of the envelope graph.

During the constraint propagation, constraints are scheduled in the order of decreasing STG of the nodes whose variables they affect. This technique permits almost loop free constraint propagation. (Some constraints may be executed more than once during the same propagation episode because of the loops in the envelope graph).

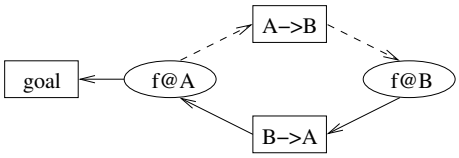


Figure 5: Envelope graph with an infinite loop.

Loop breaking. The ECT variables are lower bound estimates of the actual completion time. During construction of the envelope graph, the values of the ECT variables of leaf nodes are considered zero.

The envelope graph can contain loops, for example, when the same file `f` may be transferred back and forth between two network hosts `A` and `B`, and no job can produce this file (Figure 5). Expansion of a leaf node, e.g. `f@B` in our example, which increases the ECT of that node, may cause infinite loops in constraint propagation along graph cycles.

To avoid this, before expansion, the ECTs of all nodes reachable from the leaf node being expanded are set to infinity. Subsequent constraint propagation can only decrease the bounds, when possible. In case of an infinite loop, such as the one shown in Figure 5, the lower bounds of ECTs of unreachable nodes will remain infinity, and the constraint propagation will terminate after checking each node only once.

Solution extraction

The envelope graph is expanded until the best tree has all its leaves true in the initial state. The envelope graph is optimistic, because it can miss some interactions between different branches of the application DAG. Therefore, the best tree of the envelope graph does not necessarily represent a valid plan.

To construct a valid plan, we use critical path scheduling for the final solution extraction phase of the GPRS algorithm. A **critical path** is a path in the best tree leading from the root (the goal node) to a leaf, which chooses the most expensive support node for each AND node.

The planner **commits** nodes of the critical path starting from the leaf. Committing a node involves fixing the start time of the action and making all resource reservations belonging to the node permanent. Such commitment of reservations is possible as long as the constraint network is quiescent.

As a result of committing resource reservations, the availability of involved resources may change, which may affect other reservation constraints that involve the same resources, and, via constraint propagation, completion times of various nodes of the envelope graph. The change of ECT values of the graph nodes may change the portion of the graph considered to be the best tree and therefore result in further expansion of the envelope graph.

The current implementation of GPRS does not support backtracking. Once a node is committed, the com-

mitment cannot be revoked. In addition, when a leaf node is committed, all nodes of the critical path leading to this leaf node get frozen. The latter means that, although the values of variables of those nodes, including ECT, may change, the frozen nodes are guaranteed to be a part of the solution. Because the envelope graph is optimistic, such a non-backtracking nature of the solution extraction phase may result in suboptimal solutions. Moreover, the non-backtracking algorithm is incomplete and may fail to find a solution in the presence of budget restrictions (quotas on resource usage and/or deadlines).

On the other hand, the non-backtracking solution extraction phase is fast. Moreover, the use of the envelope graph during the solution extraction phase allows the planner to make continuous adjustments to the best tree including rescheduling actions and replacing whole subtrees. This flexibility usually leads to good quality of solutions, and appears to compensate in practice for the theoretical incompleteness of the algorithm.

Evaluation

In this section we evaluate the ability of GPRS to handle the expressiveness of the model of grid applications with explicit resource reservations and sharing, the scalability of the planner with respect to the network and application size, and the quality of solutions produced by the planner.

Handling expressiveness

A planner for computational grids needs to reason about sharing of numeric resources between jobs and data transfers running in parallel. The planner also needs to reason about action durations and start and completion times in the presence of time-varying resource availability. Finally, the planner needs to be able to select different options, such as file replicas or job types capable of producing a given data product, and trade off computation and communication so as to minimize the total duration of computation. To check if GPRS can correctly handle these tradeoffs, we ran the following experiment, which exercises the features listed above.

The abstract structure of the application is shown in Figure 6. The application consists of three jobs, organized in two levels. Each of the two jobs of the first level requires three input files, produces two output files, and takes 100 time units to complete. The third job requires four files and produces one file in 40 time units. The size of files `a`, `b`, `c`, `d`, and `e` is 500 units. Files `f`, `g`, `k`, and `q` are 100 units each. The final result file `r` has size 1000. Note that file `c` is required by two jobs, and file `k` can be produced by two different jobs. In the latter case the semantics is that the file `k` will contain exactly the same data regardless of how it was produced.

The network structure for our problem is shown in Figure 7. Replicas of several files are available on different network hosts as shown in the figure. The bandwidth of the links connecting storage nodes to the storage router is at most 5; the bandwidth of all other links is at most 10 units. We assume that only host `Server` can perform computation.

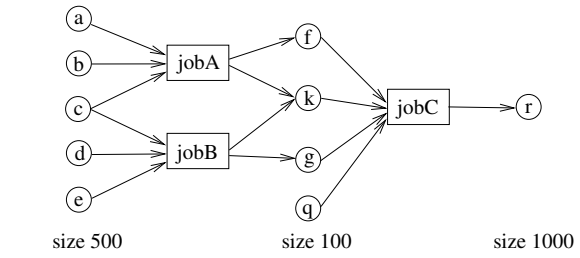


Figure 6: Abstract structure of a grid application. Circles represent files, and rectangles jobs.

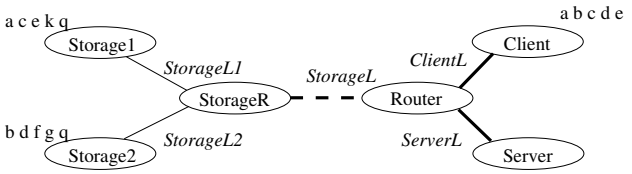


Figure 7: Network structure. Link names are shown in italics. Availability of file replicas is shown in normal font next to the hosts. Width of the lines corresponds to the link bandwidth. The dashed line shows the link, whose availability varies over time.

We further assume that the availability of the `Server` host and the availability of the link `StorageL` between the main `Router` and the storage router `StorageR` vary with time due to reservations from other ongoing computations. The availability windows for network resources are shown in Figure 8.

The goal of this problem is to obtain the `r` file on host `Client` as quickly as possible.

The plan found by GPRS is shown in Figure 9. This plan has the optimum duration given the resource availability. Note that because of the limited availability of both computational and link resources, the planner decided to recompute two of the intermediate data products and fetch the other two from where they are stored. The replica of file `q` is chosen so that the transfers of

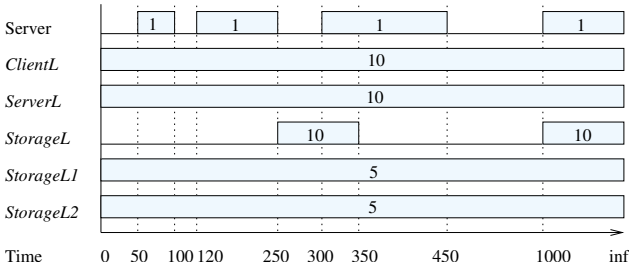


Figure 8: Resource availability. Numbers in the bars show the amount of the resources.

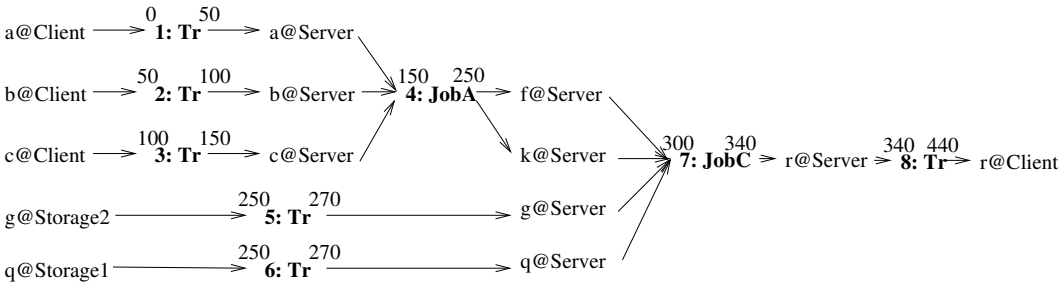


Figure 9: The final plan. File nodes describing availability of a file replica on a host are shown in normal font. Actions are shown in bold and preceded by a sequence number. The numbers above each action node are the start and end time of that action. The start time of an action depends on the earliest availability of the required files and on the resource availability.

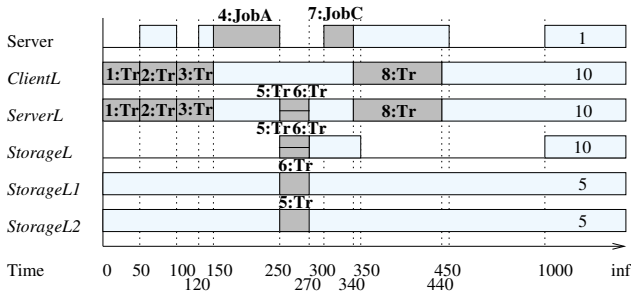


Figure 10: Resource reservations by actions. Multiple resources may be simultaneously used by a single action, and several actions can concurrently use the same resources (e.g. link *StorageL*).

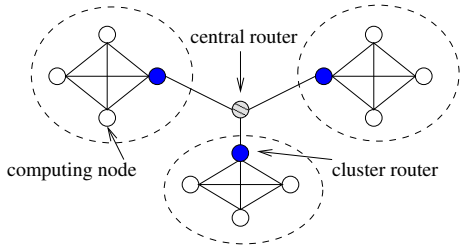


Figure 11: Synthetic network used in performance evaluation.

files `g` and `q` can be done in parallel.

Before any commitments were made, GPRS considered execution of both jobs A and B to be the fastest way to obtain intermediate files. However, committing the decision to execute job A changed availability profile of `Server`. As a result of subsequent constraint propagation, the structure of the best tree changed to include file transfers from the storage hosts.

The planner also correctly handles multiple reservations and sharing of resources (Figure 10). For example, transfer of file `g` from `Storage2` to `Server` requires simultaneous reservation of bandwidth of three links. Two of these links (`StorageL` and `ServerL`) are used for transfer of file `q` from `Storage1` at the same time.

Performance

To evaluate the scalability of the planner we used parameterized synthetic applications and networks, whose structure matches that of typical grid workflows and environments.

Figure 11 shows the network used in our experiments. This network consists of C clusters each containing N computational hosts and one router, which connects the cluster to the central master router. Hosts in a cluster can be thought of as supercomputers in a supercomputer center. Hosts within a cluster are fully connected. In total, the network contains $(N+1) \times C + 1$ hosts, of which $N \times C$ can perform computation.

Figure 12 shows the application kernel used in the experiments. The structure of this kernel is modeled after existing grid applications such as Montage (Berriman *et al.* 2003). This application kernel is parameterized, which allows us to analyze scalability of the planner with respect to different properties of the application.

The application consists of S segments limited by the splitting and merging components. The portion of the segment between these components contains W parallel execution sequences, each consisting of H processing jobs. An instance of this application contains $(H \times W + 2) \times S - 1$ jobs and $((H+1) \times W + 1) \times S$ files. The splitting and merging components serve as synchronization and data aggregation points. Jobs of a typical workflow can take tens of minutes to execute on large supercomputers, so it is feasible to invest up to several minutes of time on a workstation to construct and optimize a workflow.

In our experiments, we varied values of C, N, S, H, and W. In all cases, the goal is to achieve availability of the merged file of the last segment on the first computational host of the first cluster. Initially, all intermediate files of the first level of the first segment are available in the network. These initial files are distributed sequentially to all computing nodes of the network.

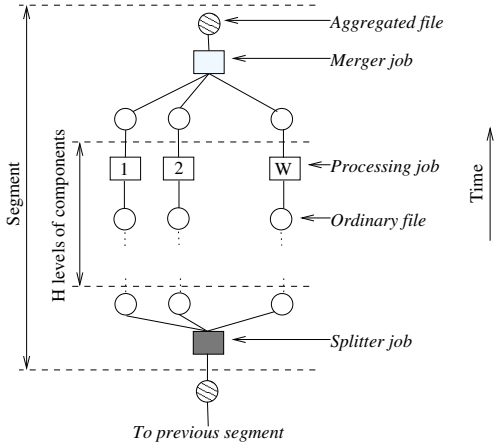


Figure 12: Application kernel.

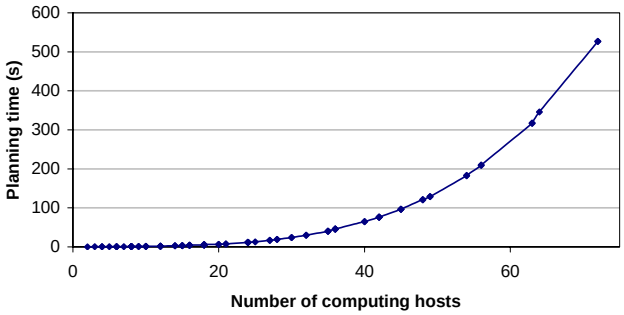


Figure 13: Scalability of GPRS wrt the network size.

Scalability with the network size. To evaluate scalability of GPRS with respect to the size of the network, we run the planner for a set of networks with parameters $C \in \{1..9\}$, $N \in \{2..9\}$ using the kernel application with $S = 1$, $H = 3$, $W = 5$. Figure 13 shows planning time as a function of the number of computing hosts in the network. As can be seen from the figure, the planning time grows fast. This can be explained by the fact that, to support simultaneous reservations of groups of network links, the planner considers all hosts of the network as targets for file transfers. The number of hosts contributes to the branching factor of the search space. The fact that the algorithm still scales to networks of considerable size can be explained by the pruning power of the envelope graph.

Scalability with the width of workflow. Next we evaluated scalability of GPRS with respect to each of the three parameters affecting the size of the workflow. Figure 14 shows planning time as a function of the total number of files in the generated workflow for the following experiments: $\{C = 2,\ N = 2,\ S = 1..36,\ H = 1,\ W = 5\}$, $\{C = 2,\ N = 2,\ S = 1,\ H = 2..100,\ W = 5\}$, $\{C = 2,\ N = 2,\ S = 1,\ H = 3,\ W = 5..125\}$.

As the results demonstrate, the planning time grows more than quadratically with the depth of the workflow. This can be explained by the fact that in the current implementation a complete constraint propagation is performed after every action choice during the solution extraction phase. The complexity of this propagation is close to linear with respect to the width of the workflow, but grows faster with respect to the depth of the workflow. We expect that a different (lazy) implementation of constraint propagation would lead to significant speedup of the algorithm. Note, however, that grid workflows tend to have few stages, and therefore scalability of the planner with respect to the depth of the workflow is less important than that with the number of parallel execution sequences.

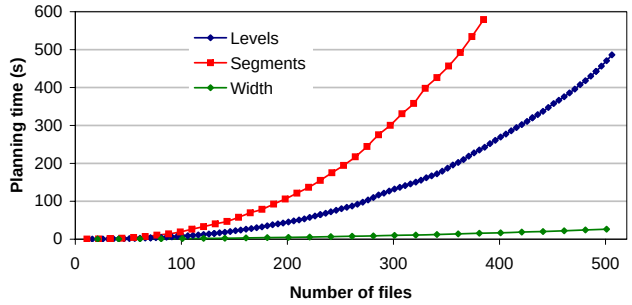


Figure 14: Scalability of GPRS wrt the size of the workflow.

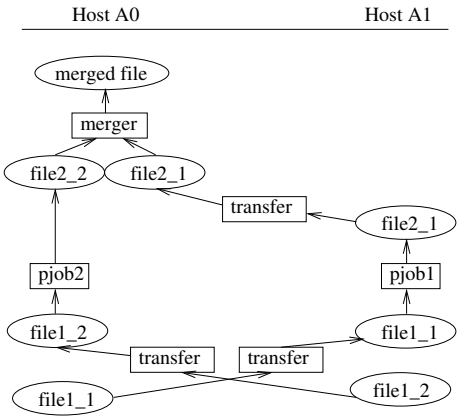


Figure 15: Suboptimal plan generated by GPRS.

Solution quality

Due to the use of the critical path scheduler with commitments for plan extraction, the plans found by GPRS may be suboptimal. For example, Figure 15 illustrates the plan produced by GPRS for the problem with $\{C = 1, N = 2, S = 1, W = 2, H = 1\}$.

The solution shown in the figure contains two file transfers more than the optimal solution. In this example, the scheduler made a greedy decision about scheduling the longest path of the workflow first, and then had to schedule the rest of the workflow using the remaining resources.

Despite possible suboptimality, GPRS still performs

good load balancing. It is problematic to find an exact optimum for problems with reasonable size. However, it is easy to check optimality of the solution in some special cases.

To assess the quality of solutions, we asked the planner to find a configuration of the kernel application described above with one segment with one job level of width 300 for a network with 2 clusters with 4 computing hosts each. We set the link bandwidths to a very high value, so that delays introduced by data transfers are negligible.

This application contains the total of 301 jobs, which can be executed on any of the 8 computing hosts. GPRS assigned 37 jobs to each of the hosts of the B cluster, 38 jobs to three hosts of A cluster (A1,A2, and A3), and 39 jobs to host A0. Since A0 is the host where the final answer was requested, the job assignment is indeed optimal.

The load-balancing effect can be explained by the fact that all decisions made by the scheduler are immediately taken into account by the envelope graph. The envelope is built over all possible execution sequences, and at any moment chooses the best way to achieve every subgoal given the current set of resource reservations.

Related work

The workflow construction problem searches for an optimum solution in the presence of complex (numeric) dependencies between actions. In designing an algorithm for solving such a problem, two decisions need to be made. First, action selection and resource allocation may be done simultaneously or separately. Second, during the search, the planner can consider each possible refinement of a plan separately, or perform aggregation using envelope-like structures.

Planners that reason about numeric resources during action selection typically support only limited form of functions (Koehler 1998; Refanidis & Vlahavas 2000). Separating these concerns allows Pegasus planner for the WCP (Blythe *et al.* 2003) to achieve good scalability. However, planners that separate action selection from resource reasoning may perform poorly in resource-driven domains (Srivastava 2000). This may be important, for example, when data transfers and job executions have comparable durations.

The second choice, single partial plan vs. envelope over multiple options, affects quality of the solution. Although single-plan algorithms may be faster, they rely on domain-specific control knowledge to guide the search (Blythe *et al.* 2003). Considering several options simultaneously helps to drive the search towards an optimum solution (Kichkaylo, Ivan, & Karamcheti 2004).

This paper presents an algorithm that uses constraint networks to construct envelopes over multiple alternative plans. This approach allows GPRS to combine the benefits of the harder options for both of the above choices with a reasonably small overhead.

Discussion and future work

Even when using a greedy critical path scheduler, GPRS still produces high quality plans. We believe that the reason for this is the use of the envelope over multiple sequences for search guidance. Every time a decision is committed to be a part of the plan, corresponding changes of the resource availability are propagated through the envelope. This may cause the change of the best tree towards the optimal solution given all committed decisions.

We believe that the planner's performance and the quality of the solution can be further improved. One possibility is to add explicit constraints between parallel execution sequences relying on the same resources. Adding such constraints will incur computational overhead. Whether or not this overhead would be justified by the improvements in the performance and solution quality is a topic for future research.

Another idea is to propagate the total resource requirements of each subtree (in addition to time) as it is done in (Kichkaylo, Ivan, & Karamcheti 2004).

Finally, it would be interesting to see how support for non-replenishable resources, e.g. quotas, which can be implemented using backtracking, affects performance of the planner.

References

Berriman, G. B. *et al.* 2003. Montage: A grid enabled image mosaic service for the national virtual observatory. In *ADASS*.

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281–300.

Blythe, J.; Deelman, E.; Gil, Y.; Kesselman, C.; Agarwal, A.; Mehta, G.; and Vahi, K. 2003. The role of planning in grid computing. In *ICAPS*.

Gerevini, A., and Serina, I. 2002. LPG: a planner based on local search for planning graphs. In *AIPS*.

Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *AIPS*.

Kichkaylo, T.; Ivan, A.; and Karamcheti, V. 2004. Sekitei: An AI planner for constrained component deployment in wide-area networks. Technical Report TR2004-851, New York University.

Koehler, J. 1998. Planning under resource constraints. In *ECAI*.

Rabideau, G.; Engelhardt, B.; and Chien, S. 2000. Using generic preferences to incrementally improve plan quality. In *AIPS*.

Refanidis, I., and Vlahavas, I. 2000. Heuristic planning with resources. In *ECAI*.

Smith, D.; Frank, J.; and Jónsson, A. 2000. Bridging the gap between planning and scheduling. *Knowledge Engineering Review* 15(1).

Srivastava, B. 2000. Realplan: Decoupling causal and resource reasoning in planning. In *AAAI*.

Scheduling a plan in time: a CSP approach

Eliseo Marzal, Eva Onaindia and Laura Sebastiá

Departamento de Sistemas Informáticos y Computación,
Universidad Politécnica de Valencia, Spain
{emarzal,onaindia,lstarin}@dsic.upv.es

Abstract

In temporal planning domains, the exact duration of actions are usually known at planning time so a temporal planner is able to compute the best plan w.r.t makespan. However, in many realistic domains the exact duration of actions is only known at the instant of executing the action. This is the case, for instance, of temporal domains where it is common to find external factors that cause a delay in the execution of actions. In this case, the actual duration of a temporal action depends on the timing at which the action is actually executed.

In this paper we present an approach to obtain a plan for a temporal domain with delays. Our approach consists in combining a planning process, from which a temporal plan is obtained, and a scheduling process to allocate (instantiate) such a temporal plan over a time line. This latter process takes into account the delays defined in the domain while allocating actions in time.

Introduction

Research in AI planning is more and more concerned with the introduction of more expressive languages and development of new techniques to deal with more realistic problems. A crucial element in this approach to reality is time. In the last years, several extensions in the standard language PDDL have represented a step forward in the resolution of temporal planning problems, as the introduction of durative actions in PDDL2.1(Fox & Long 2003) or timed initial literals in the most recent PDDL2.2(Edelkamp & Hoffmann 2004).

However, handling time in real planning problems is much more than dealing with durative actions or incorporating temporal constraints. Time plays an important role because it is a source of imprecision and uncertainty. In this direction we can find in the literature several approaches to handle an extended model of durative actions, as the introduction of uncertainty in the time consumption of actions ((Biundo, Holzer, & Schattenberg 2004), (Bresina *et al.* 2002)).

Time also plays an important role when plans are to be executed in a specific temporal setting. The user might want the plan not to finish later than time t, or start at time t' as earliest or to have a maximum duration d. These constraints may be considered during planning if the planner is able to handle them and the language used for domain modelling accounts for it.

More relevant are the intrinsic restrictions to the particular domain or environment. In this category we can find the typical delays that affect the execution of actions. Actions do not always take the same time to execute but real durations depend on the timing at which actions are actually executed, which may be affected by a longer or shorter delay. A lot of elements condition the real duration of actions: moving from one place to another is more costly at peak hours, loading objects in a container depends on the number of resources (hoists, humans) available when the load is carried out and a space operation might last differently whether it is carried out in daytime or at night.

In this paper we present an approach to obtain a plan for temporal domains with delays. The approach integrates a standard planning process, from which a temporal plan is obtained, with a CSP resolution to instantiate the temporal plan over a real time line. Our work brings two main contributions. First it introduces a model to handle delays in temporal planning domains, a concept that has not been previously dealt with in the literature. Second we present a different way of handling actions with variable durations through a scheduling process rather than within the planning process.

The paper is organized as follows. The following section motivates the use of delays in temporal environments and introduces some concepts to define action delays. Section *Planning in temporal domains with delays* sketches the overall working schema of our approach. Section *Applying delays* presents the algorithm used to apply delays to a given temporal plan. Section *Plan scheduling as a CSP* outlines the steps to encode the plan information and fulfil the temporal requirements specified by the user. Section *Experimental Results* presents some results that show different schedulings of a temporal plan over a time line and how this information can be exploited to satisfy the time restrictions. Finally, the last section concludes and outlines some future work.

Delays in a temporal setting

In this section we give an exposition to motivate the introduction of action delays in temporal planning. The key issue in the proposed approach is that actions or activities seldom

last as initially planned (expected) in daily life.

Usually, when we refer to a "delayed action" we mean the action execution does not finish at the scheduled time. There are several reasons that can be a cause of delay:

- the action execution does not start at the expected time and therefore it does not finish at the scheduled time, i.e. the action is delayed due to a later happening of the starting time. In this case there are no variations in the duration and the action is simply shifted forward in time. These delays are usually produced by the causal relationships the action holds with other actions that, in turn, are also delayed (a chain-delay effect).
- the action execution starts at its scheduled time but the duration is longer than expected. These delays are normally caused by external factors that affect the action execution. Although these external conditions can be predicted in advance they are not static causes that happen forever but only rather at specific times along the action execution.
- both reasons stated in the above items. Delays in the action duration give rise to a later ending time and, consequently, dependent actions start later and so on.

In PDDL 2.1, the duration of the actions is dependent on the particular action parameters. For example, in a classical logistics or transportation domain these parameters can be the origin, the destination, the truck, the driver etc. Variable durations can be modelled through the use of functions (level 3 durative actions).

Let's now add one more dose of reality. We all know that driving at night is harder than driving during daytime; driving at peak-hours is slower than driving at regular hours; driving under the rain is also slower; a load/unload operation that is carried out at lunch time will take longer as the number of available resources is half the usual. Under all these situations the standard action duration (under normal conditions) is increased by external factors to the action itself.

In principle, delays are produced by generic causes that affect action durations equally, no matter the timing at which actions are executed. For instance, driving with a heavy traffic flow implies to move slower, regardless if it happens at 9 a.m. or at 9 p.m..

However, many common causes of delay typically occur at particular times or time intervals. Peak-hours usually occur early in the morning or late in the afternoon after work. A lack of human resources may likely occur at lunch time. Therefore, an action may be affected by different delays along its execution depending on the time intervals over which the action is executed. If action `drive-truck T1 A B` starts at 8 a.m. and it takes two hours under normal conditions, the first driving hour may be affected by a heavier traffic flow than the second hour. Unloading a truck that usually takes one hour may be longer because the action starts at 11.30 a.m. and there are fewer workers from 12.00 a.m. on.

A delay is the extra-time to put in the action duration over a time interval due to an external factor. Let's define $\alpha_{<cause>}$ the delay caused by any reason. Figure 1 shows two driving actions affected by different delays at different times. Road connecting `B` to `C` is a highway so heavy traffic never occurs. And road connecting `A` to `B` is located in the west part so darkness comes later.

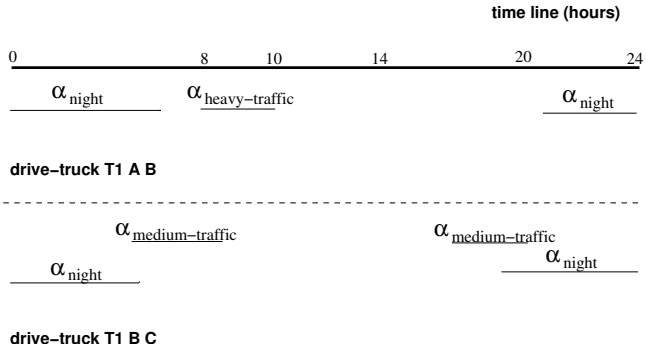


Figure 1: Causes of delays in two drive actions

The value of $\alpha_{<cause>}$ can be specified as:

1) a fixed value (i.e. 5 minutes)
2) a value proportional to the standard action duration (i.e. 5% of the standard duration or equivalently a fraction 5/100 in the interval [0,1])
3) a value given by a formula (i.e. $\mathsf{speed} * \mathsf{weight} - \mathsf{distance}$)

Additionally, we can distinguish between causes of delays that are known in advance and dynamic causes that appear during the plan execution and cannot be predicted at planning time. In the first group we can find: the traffic flow in the roads, the existance of works in the roads, the happening of a mass event in the city, a shortage of workers, the night fall, etc. Among the second group we can find: the appearance of rain, traffic-lights stop working and any kind of unexpected event. In the rest of the paper we will focus just on the first group of delays.

Delays over time

As we mentioned in previous section, the value of $\alpha_{<cause>}$ can be specific for each action if $\alpha_{<cause>}$ is defined in terms of some of the action parameters. We will denote by $\alpha_{<cause,a_i>}$ the value that results from computing $\alpha_{<cause>}$ for a_i. $\alpha_{<cause,a_i>}$ represents the increase fraction that has to be applied on a_i when the $< cause >$ of delay is found during the execution of a_i.

The total delay of an action will depend on its start time of execution so the same action will have different delays at different starting execution times. In order to compute the delay of an action we have first to define the delays that affect the action over the time line. Thus, it is necessary to represent the *delay patterns* of actions over a time window. Typically, the range of the time window would be the overall plan duration but it is also possible to specify a repetitive delay pattern over sequential time windows along the plan duration. In this paper we study a general case where the time window covers the range of a whole day from 0 to 24

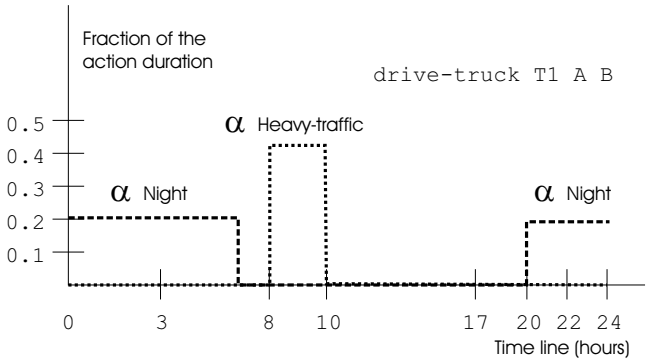


Figure 2: Graphical representation of delays

hours but the analysis can also be applied to a different temporal scope.

For each action a_i and time window $[t_b, t_e]$ (in our case $t_b = 0, t_e = 24$) we define a set of subintervals over $[t_b, t_e]$ and specify a delay associated to each of them. This set of subintervals along with their corresponding delays are called *timing intervals* of an action a_i. When a_i is executed over a timing interval, the actual duration of a_i is augmented with the delays indicated in the timing interval.

There can be more than one cause of delay over a timing interval. We denote by $\alpha_{c1[a_i,t_i,t_j]}$, $\alpha_{c2[a_i,t_i,t_j]}$, $\ldots$, $\alpha_{cn[a_i,t_i,t_j]}$ the values of the delays produced by causes $c_1, c_2, \ldots, c_n$ on action a_i over timimg interval $[t_i, t_j]$. That is, the result of $\alpha_{<c_k,a_i>}\ \forall c_k \in < cause >$ over timing interval $[t_i, t_j]$. For the sake of simplicity, we will denote by $\alpha_{[a_i,t_i,t_j]}$ how long action a_i is totally delayed over $[t_i, t_j]$ due to all the delay causes produced over such a timing interval.

The definition of delays over timing intervals for an action a_i can be interpreted as a continuous function over the time line. This scheme would also allow to specify delays as a probability distribution. Figure 2 shows the timing intervals of the action `drive-truck T1 A B` for $\alpha_{heavy-traffic}$ and α_{night}. Thus, heavy traffic only affects timing interval $[8, 10]$ and the action is delayed $0.4 \times dur(a_i)$; and driving at night produces a retard over the timing interval $[20, 7]$ of $0.2 \times dur(a_i)$.

Planning in temporal domains with delays

In PDDL, the duration of an action in a temporal domain can be expressed as a fixed value (simple time) or the result of a numeric-valued fluent (time). The straightforward consequence of handling temporal domains with delays is that the same action can have different durations depending on the timing intervals over which it is executed, that is the duration will be different depending on its start execution timing.

Dealing with delays when a temporal plan is being computed adds a great complexity to the planning process because the actual duration of each action is unknown until it is allocated in the time line. One alternative would be to compute in advance the duration of each single action at each time point. However, although this can be done in polynomial time, the result would be a prohibitive number of different instantiated actions (one instance for each different possible duration) which would cause a blow up in the planning process. Moreover, computing accurate estimations of the plan duration to define heuristics would be much harder as the exact duration of actions is unknown.

Therefore, our proposal is not to consider the allocation (instantiation) of a plan in time into the temporal planning process but in a separate post-planning process (see figure 3). First, we obtain a plan from an existing temporal planner and the start/end time of the plan is initially set to a value on the real time line, according to the user specifications; otherwise, the start time is set to any value. Then we calculate the extended duration of the plan when delays are applied. Once we have the plan allocated in time, we check if the user temporal constraints hold in the plan. If this is the case, a solution is returned. Otherwise, the plan is converted into a CSP and its resolution will provide new allocations of the plan in time until we find a solution that satisfies the user restrictions.

The post-planning process performs two main tasks:

- **Apply delays to a temporal plan**. When the plan is set on a time line and, consequently, the timing intervals over which actions are executed are known, an algorithm to compute the extended duration of the plan is applied. This is explained in detail in next section.
- **Scheduling a plan as a CSP**. The objective of converting a plan into a CSP is to find new allocations in time for the plan. Actions are reordered and re-allocated in time so as to minimize the overall delay and satisfy the user restrictions. This is explained in section *Plan scheduling as a CSP*

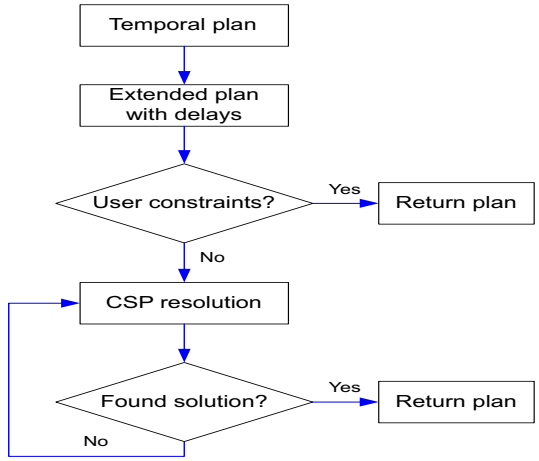


Figure 3: Post-planning process to compute a plan in a temporal domain with delays

Applying delays

In this section we present the algorithm to compute the extended duration of a temporal plan given the action delays at

$p = 0$
do
if $(d_k + d_k \times \alpha_{[a_k,t_p,t_{p+1}]} \leq (t_{p+1} - \max(t_p, beg_k))$
then
$end_k = \max(t_p, beg_k) + d_k + d_k \times \alpha_{[a_k,t_p,t_{p+1}]}$
else
$\Pi = (t_{p+1} - \max(t_p, beg_k))/(1 + \alpha_{[a_k,t_p,t_{p+1}]})$
$d_k = d_k - \Pi$
endif
$p = p + 1$
while $d_k > 0$

Figure 4: Algorithm for delays application

their timing intervals. The extended (real) duration of a parallel plan P composed of several sequences of actions will be the extended duration of the longest sequence.

We initially set the start/end time of the plan according to the user specifications. This gives us information about the timing intervals over which actions will be executed and we can then proceed to compute the extended duration of the plan. We show here how to compute the extended duration of a single action; the same process is then repeated for every action and the final extended plan duration is calculated as indicated above.

Let a_k be a durative action which start time and standard duration are denoted by beg_k and d_k respectively. a_k starts at timing interval $[t_0, t_1]$ and finishes at timing interval $[t_{n-1}, t_n]$ (see case 0 in Figure 5). The basic procedure of the algorithm works as follows: for action a_k and timing interval $[t_i, t_j]$, find out "how much" of the duration of a_k over $[t_i, t_j]$ actually corresponds to the action execution and "how much" corresponds to causes of delays in the timing interval; then substract the action duration from d_k and repeat again for the next timing interval. Figure 4 shows the algorithm for applying delays to a single action a_k.

The expression $(t_{p+1} - \max(t_p, beg_k))$ in the if condition always returns the range of timing interval $[t_p, t_{p+1}]$ except the first time if a_k does not start exactly at t_p. The expression $(d_k + d_k \times \alpha_{[a_k,t_p,t_{p+1}]})$ denotes the extended duration of a_k over timing interval $[t_p, t_{p+1}]$. If this duration goes beyond t_{p+1} then we compute "how much" of $(t_{p+1} - \max(t_p, beg_k))$ (duration of a_k over $[t_p, t_{p+1}]$) actually corresponds to the action execution (Π), such that $\Pi + \Pi \times \alpha_{[a_k,t_p,t_{p+1}]} = t_{p+1} - \max(t_p, beg_k)$ (cases (1), (2), (3) and (4) in figure 5). d_k is then updated and the process is repeated for the next timing interval while $d_k > 0$. If the extended duration completely falls within $[t_p, t_{p+1}]$ (case 5 in figure 5) then end_k is updated by adding this extended duration to the range of the previous timing intervals (case 6 in figure 5). The final result will be an extended a_k duration, as it can be observed in case 6 of figure 5.

Plan scheduling as a CSP

This step is executed when the scheduled plan P does not fulfil the user requirements. We build a CSP that encodes:

- information in plan P

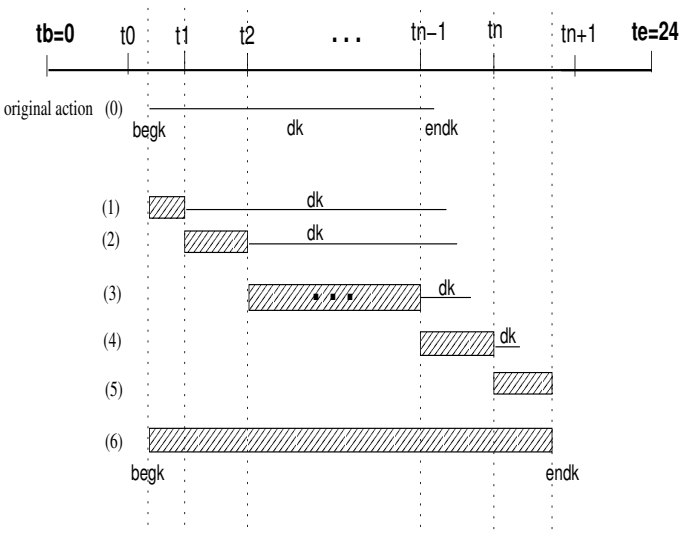


Figure 5: Graphical representation of delays application

- all possible planning alternatives in P (different ways of achieving a literal with actions in P)
- the different durations actions in P can take on according to their timing intervals
- user constraints

It is important to remark that, unlike other approaches ((Do & Kambhampati 2001), (Kautz 2004)) our CSP does not encode the planning problem but a scheduled plan in time plus all the necessary information to modify the plan and make it accomplish the user requirements.

CSP specification

Let P be a partially-ordered set of actions $\{a_o, a_1, \ldots, a_n\}$; a_0 and a_n are the initial and final ficticious actions that represent the start and end time of P, respectively. We will use s_i and e_i to denote the start and end time of an action a_i, and t_i to refer either s_i or e_i indistinctly. Let X be the set of variables $X = \bigcup_{i=0}^{n} s_i \cup \bigcup_{i=0}^{n} e_i$. For the ficticious intial and final actions, $s_0 = e_0$ and $s_n = e_n$. Time is modelled by $\mathcal{R}^+$ and their chronological order. Particularly, the list of possible values for an end time point is $D_{e_i} = \{0 \ldots 24\} \in \mathcal{R}^+$ and for a start time point is $D_{s_i} = \{[0, v_1][v_1, v_2] \ldots [v_n, 24]\} \in \mathcal{R}^+$ where v_i denote the left/right extreme points of the timing intervals. We distinguish three types of constraints C: planning constraints, timing interval constraints and user constraints.

This specification gives rise to a TCSP (Ghallab, Nau, & Traverso 2004) as the problem involves a set of variables X having continuous domains ($\mathcal{R}^+$), each variable represent a time point and the three types of constraints are encoded as a set of unary and binary constraints (see below).

(A) Planning constraints. We show here how to encode planning relationships into temporal constraints.

- *Causal link*. We denote a causal link (Penberthy & Weld

1992) between actions a_i and a_j as (a_i, a_j, p) where p is the literal that a_i produces for a_j. The translation of a causal link into a temporal constraint has four variants, depending on literal p is produced at start or end of a_i and required at the start or end of a_j. These four variants are encoded as binary constraints: $s_i \leq s_j$, $s_i \leq e_j$, $e_i \leq s_j$ or $e_i \leq e_j$. We will use the general temporal constraint $t_i \leq t_j$ to refer to any of the four variants.

The different planning alternatives (ways of producing literal p with actions in P) are encoded as a disjunctive temporal constraint:

$$\bigvee_{\forall a_k / p \in add(a_k)} t_k \leq t_j \quad (1)$$

- *Threat*. Let's suppose a_k threatens causal link (a_i, a_j, p). The set of disjunctive constraints to model the demotion and promotion choices are:

$$\bigwedge_{\forall a_k / p \in del(a_k)} t_k < t_i \vee t_k > t_j \quad (2)$$

In the case of an *overall* condition the disjunctive constraint would be $\bigwedge_{\forall a_k / p \in del(a_k)} t_k < t_i \vee t_k > e_j$.

We classify planning constraints into three different categories:

1. *Safe causal link*. When a_i is the only way to produce literal p and there is no action, except may be a_j, that deletes p. This implies a_i is the only producer so the causal link is unmodifiable and there are no threats over (a_i, a_j, p). Therefore, a safe causal link is encoded as temporal constraint that represents a causal link.
2. *Hard causal links*. When a_i is the only way to produce p and it exists at least one action that deletes p. This implies a_i is the only producer so the causal link is unmodifiable and there can be threats over (a_i, a_j, p). Therefore, a hard causal link is encoded as a temporal constraint that represents the causal link plus a set of disjuntive constraints to avoid threats over such a causal link.
3. *Weak causal links*. When there are several producer actions of p for a_j. A weak causal link is encoded by using the disjunctive temporal constraint of the planning alternatives plus a set of disjuntive constraints to avoid threats on each possible causal link. The combination of these contraints is encoded as:

$$\bigvee_{\forall a_i / p \in add(a_i)} t_i \leq t_j \wedge (\forall a_k / p \in del(a_k)(t_k < t_i \vee t_k > t_j)) \quad (3)$$

(B) Timing intervals constraints. These constraints represent the different durations an action a_i can take on depending on the timing interval where a_i starts its execution. For example, assuming we have $\alpha[a_i, 3, 5]$ and $\alpha[a_i, 5, 9]$, the final duration of a_i will be different if $s_i \in [3, 5]$ or $s_i \in [5, 9]$. Actually, the final duration also depends on the specific time point within the corresponding timing interval but we will simplify the modelling by just assuming that the extended duration of a_i is the same when s_i falls at any point within the same timing interval[1]. The timing interval constraints are encoded as:

$$\forall a_i \in P \bigvee_{\forall [t_p, t_{p+1}] \in [t_b, t_e]} t_p \leq s_i \leq t_{p+1} \wedge e_i = s_i + d \quad (4)$$

where d is calculated by applying the algorithm of figure 4. We restrict the plan to be executed before $t_e = 24$.

(C) User constraints. Constraints on the start/end time of the plan are expressed as $s_0 \geq v_0$, $e_n \leq v_n$. The same specification is used to represent restrictions on the start/end time of the actions in the plan ($t_i \leq v_i$ $t_i \geq v_i$). A limit in the plan duration is encoded as $e_n = s_0 + d_k$.

Example. Figure 6 represents a scheduled plan in the interval [10,14:30]. It shows a plan where each action is represented by a rectangle with its preconditions (Pr) and effects (Ef). The causal links between the actions are denoted by arrows with labels on the form (type of causal link, literal). All conditions are at start except condition v of a_4 which is overall and all add and del effects are at end.

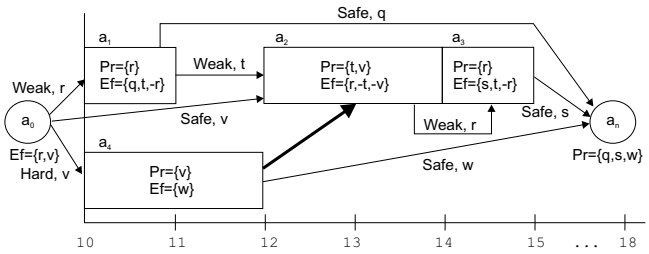


Figure 6: A scheduled plan over a time line

The set of temporal constraints representing the planning relationships are:

- Safe causal links: $e_0 \leq s_2$, $e_1 \leq s_n$, $e_3 \leq s_n$, $e_4 \leq s_n$
- Hard causal links: $e_0 \leq s_4$, $e_2 > e_4$ (this latter constraint represents the only way to avoid the threat of a_2 over the hard causal link)
- Weak causal links: Let's take, for instance, the weak causal link (a_0, a_1, r). The backup producer is a_2 so the planning alternatives are $e_0 \leq s_1 \vee e_2 \leq s_1$. The possible threatener is a_3. Consequently, the final encoding for this weak causal link implies one way of solving a threat over the first planning choice and the two ways of avoiding a potential threat on the second planning choice:
  $[e_0 \leq s_1 \wedge (e_3 > s_1)] \vee [e_2 \leq s_1 \wedge (e_3 < s_2 \vee e_3 > s_1)]$
  Similarly for the rest of weak causal links encodings.

The set of temporal constraints to represent the timing intervals and user restrictions are:

- $s_0 \geq 10$, $e_n \leq 14:30$ (user constraints)

[1] In this first aproximation we only consider the left extreme of timing intervals. In a future work, we will extend this approach to work with the exact duration of each action according to the time point within the timing interval.

- $(0 \leq s_1 \leq 12 \wedge e_1 = s_1 + 1) \vee (12 \leq s_1 \leq 24 \wedge e_1 = s_1 + 2)$
- $(0 \leq s_2 \leq 6 \wedge e_2 = s_2 + 1.5) \vee (6 \leq s_2 \leq 14 \wedge e_2 = s_2 + 2) \vee (14 \leq s_2 \leq 24 \wedge e_2 = s_2 + 3)$
- $(0 \leq s_3 \leq 10 \wedge e_3 = s_3 + 2) \vee (10 \leq s_3 \leq 18 \wedge e_3 = s_3 + 1) \vee (18 \leq s_3 \leq 24 \wedge e_3 = s_3 + 4)$
- $0 \leq s_4 \leq 24$

Solving the CSP

After encoding the current plan into a TCSP, we have a representation of this plan joint with all the possible alternative plans we could build. The next step is to solve this TCSP in order to obtain a new plan, which satisfies the user requirements. As shown above, this is a disjunctive TCSP, with unary and binary constraints. A straightforward way of solving it is to decompose it into several non-disjunctive TCSP (Detcher 2003). This is the approach we have adopted, but we need to distinguish between two levels of decomposition: the first level considers the disjunctive planning constraints (weak and hard) while the second level considers the disjuntive timing interval constraints. Once a non-disjuntive TCSP has been built, we apply an arc-consistency procedure in order to shrink intervals domain for each s_i. The remainder of this section formalizes the algorithm to solve the general TCSP.

First, we need to define the intersection of intervals (Detcher 2003). Let $T = \{I_1, \ldots, I_l\}$ and $S = \{J_1, \ldots, J_m\}$ be two constraints representing the domain of intervals of a temporal variable s_i or e_i. The *intersection* of T and S, denoted by $T \oplus S$, admits only values that are allowed by both T and S, that is, $T \oplus S = \{K_1, \ldots, K_n\}$ where $K_k = I_i \cap J_j$ for some i and j.

Our algorithm works in four stages. At any time, the algorithm can return that there is no solution when any of the variable domains becomes empty. These stages are:

1. **Dealing with the user requirements.** The first step of this algorithm consists in shrinking the given domains of the variables according to the user requirements. We distinguish three types of user requirements and each of them has a different process:

   - *Start and end of the plan:* Given a start and end of the plan constraints of the form $s_0 \geq v_0$ and $e_n \leq v_n$, we can shrink the domains of all the variables in the CSP as we implicitly know that $\forall a_i \in P, s_0 \leq s_i \wedge e_i \leq e_n$. Given that D_{t_i} represents the domain of a variable s_i or e_i, the new domains are computed as follows:

     $$D_{t_i} = D_{t_i} \oplus [v_0, v_n]$$

     In case s_0 or e_n are not restricted, then it is assumed that $v_0 = -\infty$ and $v_n = \infty$.
   - *Duration of the plan:* This requirement cannot be used to shrink the variable domains, because it does not restrict at what time an action may or may not start. It only shrinks the makespan of the plan which is not represented in the variable domains.
   - *Start and end of one action:* Given two constraints indicating the start and end of an action of the form $s_i \geq v_0$ and $e_i \leq v_n$, we can shrink the domain of these variables in the same way as with the start and end of the plan.

2. **Selection of the planning disjunction.** We focus on those constraints which produce disjunctions: hard and weak causal link constraints. Let $CL = (a_i, a_j, p)$ be a causal link.

   - Assuming that CL is a hard causal link, let c_j be a threat constraint related with CL on the form of (2). Therefore, the number of different TCSP we can build considering only CL is $NH_j = 2^{|a_k|}$, where $|a_k|$ is the number of actions that threaten CL.
   - Assuming that CL is a weak causal link, let c_j be the constraint corresponding to CL on the form of (3). Therefore, the number of different TCSP we can build considering only CL is $NW_j = |a_i| \times 2^{|a_k|}$, where $|a_i|$ is the number of actions that can solve p for a_j and $|a_k|$ is the number of actions that threaten CL.

   The number of different TCSP grows exponentially as the number of hard and weak causal links increases. Namely, this number is:

   $$\prod_{c_i \in HCL} c_i \times \prod_{c_i \in WCL} c_i$$

   where HCL and WCL are the set of hard and weak causal link constraints, respectively. We select one of the obtained TCSP randomly[2].

3. **Selection of the timing interval disjunction.** At this moment, we only consider the TCSP selected in the previous step. Let $c_i = \bigvee_{\forall [t_p, t_{p+1}]} (t_p \leq s_i \leq t_{p+1} \wedge e_i = s_i + d)$ be a timing interval constraint. Again, we can build a number of different TCSP, each of them considering a different execution interval for each action. This number is $\Pi_{\forall a_i \in P} |N_{Ia_i}|$, where $|N_{Ia_i}|$ is the number of timing intervals for each action. Fortunately, this number is an upper bound of the number of the TCSP we must actually consider, as some of these combinations rule out due to the domains restriction performed in previous steps. We select a TCSP which will be solved in the next step.

4. **Arc-consistency.** An arc-consistency process is applied in order to shrink the domain for each variable, that is, we do not calculate the minimal domain because the start/end time points of the actions come conditioned by the start/end time points of the previous actions. The arc-consistency process we have implemented takes $O(n^2)$, where n is the number of variables.

Example We continue with the example of the previous section. The domains of the variables are those represented in the timing interval constraints. Taking into account the user requirements, the new domains of the variables after applying the first step of the TCSP resolution process will be:

[2] At this moment, there is no reasoning about which of the obtained TCSP may lead to a solution with a higher probability. This will be discussed in the experiments section.

Timing intervals	Duration <	**300**	**250**	**200**
2	Makespan	238	238	No plan
	Start-End time	0:00-3:58	0:00-3:58	
	Time (secs.)	0.031	0.031	2.14 (4.48)
Timing intervals	Duration <	**300**	**260**	**245**
4	Makespan	276	258	
	Start-End time	0:00-4:36	17:18-21:36	
	Time (secs.)	0.047	543	Time Out
Timing intervals	Duration <	**300**	**250**	**200**
6	Makespan	222	222	
	Start-End time	0:00-3:42	0:00-3:42	
	Time (secs.)	0.016	0.031	Time Out

Table 1: Duration constraints

Timing intervals	Start-End time	**[10,20]**	**[12,20]**	**[14,19]**	**[14,18:35]**
2	Makespan	245	264	284	No plan (274)
	Start time	10:00	12:00	14:00	(14:00)
	End time	14:05	16:24	18:44	(18:34)
	Time (secs.)	0.17	0.047	0.031	0.031 (0.0625)
Timing intervals	Start-End time	**[10,20]**	**[12,20]**	**[14,19]**	**[14,18:20]**
4	Makespan	301	296	268	No plan (253)
	Start time	10:00	12:00	14:00	(14:00)
	End time	15:01	16:56	18:28	(18:13)
	Time (secs.)	0.843	0.0625	0.0468	0.172 (0.906)
Timing intervals	Start-End time	**[10,20]**	**[12,20]**	**[14,19]**	**[14,18:30]**
6	Makespan	284	286	273	No plan (263)
	Start time	10:00	12:00	14:00	(14:00)
	End time	14:44	16:46	18:33	(18:23)
	Time (secs.)	0.25	0.0624	0.203	2.14 (3.46)

Table 2: Constraints on the start and end of the plan

- $D_{s_0} = D_{e_0} = \{[10, 14 : 30]\}$
- $D_{s_1} = \{[10, 12], [12, 14 : 30]\}, D_{e_1} = \{[10, 14 : 30]\}$
- $D_{s_2} = \{[10, 12], [12, 14 : 30]\}, D_{e_2} = \{[10, 14 : 30]\}$
- $D_{s_3} = \{[10, 14 : 30]\}, D_{e_3} = \{[10, 14 : 30]\}$
- $D_{s_4} = \{[10, 14 : 30]\}, D_{e_4} = \{[10, 14 : 30]\}$
- $D_{s_n} = D_{e_n} = \{[10, 14 : 30]\}$

As shown in the previous section, there is a number of hard and weak causal links, which in turn define different TCSP. We select the following planning disjunctions: $e_3 \leq s_2$, $e_2 \leq s_1 \wedge e_3 < e_2$ and $e_0 \leq s_3 \wedge e_1 > s_3$.

Now, we have to select a set of disjunctions from the timing interval constraints. Let's assume we select the following ones: $(0 \leq s_1 \leq 12 \wedge e_1 = s_1 + 1)$, $(0 \leq s_2 \leq 6 \wedge e_2 = s_2 + 1.5)$, $(0 \leq s_3 \leq 10 \wedge e_3 = s_3 + 2)$ and $0 \leq s_4 \leq 24$.

In this case, it is obvious that the obtained TCSP is inconsistent, as constraint 2 cannot be satisfied ($D_{s_2} = \{[10, 12], [12, 14 : 30]\}$). If instead of disjunction $(0 \leq s_2 \leq 6 \wedge e_2 = s_2 + 1.5)$, we select disjunction $(6 \leq s_2 \leq 12 \wedge e_2 = s_2 + 2)$, the TCSP is consistent and after applying the arc-consistency, we obtain the following start time points for each action: $s_0 = 10, s_1 = 13, s_2 = 11, s_3 = 10, s_4 = 10, e_n = 14$. Therefore, this new plan fulfil the user requirements.

Experimental Results

In order to check the behaviour of our TCSP, we run different experiments with a hand-made problem from the `driverlog` domain[3]. The temporal plan returned by LPG (Gerevini, Saetti, & Serina 2004) for this problem instance contains 18 actions and a makespan of 200 time units (before applying delays).

The above tables represent different temporal restrictions specified by the user for three different temporal settings (2, 4 and 6 timing intervals). The first row in the results for each timing interval represents the plan makespan, the second row the start and end time of the plan and the third row the CPU time in seconds. All experiments were run on Pentium IV 3GHz with 512 Mb of memory and censored after 15 minutes.

Table 1 shows the results when the user requirement is to set the plan duration below three different values (300, 250 and 200 time units). For 2 timing intervals and a duration of less than 200 seconds no plan is found. The value in brackets indicates a second CSP has been solved with no success.

Table 2 shows the results when the user imposes restrictions on the start and end time of the plan within a time interval. For example, first column indicate that the start and end time of the plan must fall within the interval [10,20]. We can see that the most time-consuming experiment is for the longest plan execution interval. This clearly shows that when the plan execution interval is more restricted, the number of consistent timing intervals decreases and the search is faster. However, in this case, it does not exist a valid combination of timing intervals, the necessary time to return *No plan* increases as long as the number of timing intervals. Again values in brackets represent the results obtained from a second CSP resolution.

Table 3 represents a restriction on the plan duration plus a restriction on the start time of the plan. First column denote that the makespan must be below 250 seconds and the start time of the plan at any point later than time 10.00. The missing data for the start/end time rows correpond to those cases where a plan is not found.

We can also observe that CPU time is shorter in Table 1 than Table 3. In general, if only restrictions on the duration are considered then appropriate timing intervals combinations are found faster. In the case of 4 timing intervals and a duration below 260 time units we notice the solution found (258 t.u.) is very close to the limit and the CPU time is very high (543 seconds) which indicates that almost all possible combinations of timing intervals have been searched until finding a solution. From comparing results in Table 2 and Table 3 we can deduce that when there are restrictions on the start and end time of the plan the computation is faster

[3]Domain from the International Planning Competition 2002: http//planning.cis.strath.ac.uk/competition

Timing intervals	Duration / Start time	**< 250 / >10**	**<240 / >10**	**<300 / >12**	**<250 / >12**			
2	Makespan	245	No plan (No plan)	264	No plan (No plan)			
	Start-End time	10:00-14:05		12:00-16:24				
	Time (secs.)	0.17	2.25 (4.46)	0.0156	0.01562 (0.078)			
Timing intervals	Duration / Start time	**<300 / >10**	**<300 / >12**	**<300 / >14**	**<260 / >14**	**<300 / >18**	**<230 / >18**	
4	Makespan	291	286	268	258	240	No plan (No plan)	
	Start-End time	10:00-14:51	12:00-16:46	14:00-18:28	17:18-21:36	18:00-22:00		
	Time (secs.)	2.2	0.0426	0.0316	0.0624	0.0171	0.078 (0.185)	
Timing intervals	Duration / Start time	**<300 / >10**	**<300 / >12**	**<300 / >14**	**<250 / >14**	**<300 / >18**	**<220 / >18**	**<200 / >18**
6	Makespan	284	286	273	249	228	219	No plan (No plan)
	Start-End time	10:00-14:44	12:00-16:46	14:00-18:33	17:13-21:22	18:00-21:49	18:59-22:38	
	Time (secs.)	0.87	6.62	3.49	3.73	0.156	0.3124	2.125 (4.2)

Table 3: Constraints on the duration and the start time of the plan

because it is possible to rule out many more combinations of timing intervals.

In general, a conclusion from the experiments is that the more number of timing intervals the longer time for the CSP resolution. When dealing with 2 timing intervals, the number of possible action instantiations is about 4000 and for 4 timing intervals is about 16 million of combinations. The CPU time of the experiments depend on when the intervals that provide the shortest duration to actions are selected in the CSP resolution, the set of valid intervals according to the start/end or duration restriction, etc. As a whole, the most time-consuming experiments are those with the earliest start time and shortest makespan.

As a conclusion, what we want to highlight from the experiments is that the CPU time for the CSP resolution is perfectly affordable even though no heuristic information at all has been used to select the most appropriate CSP or timing intervals for a particular solution. Moreover, modeling time restrictions at execution time is much simpler in a CSP framework than in a planning system.

Conclusions and further work

In this paper we have presented an extended model of durative actions which takes into account the fact that actions are delayed at the time of being executed in a real domain. This way, we introduce the concept of delay as the increase in the duration of an action due to very common causes in daily life but rarely considered in planning modelling. The introduction of delays create a complete different scenario in temporal planning. Now plans may have a different makespan according to its timing of execution and this aspect has to be taken into account in order to meet the user restrictions on the start/end time or duration of the temporal plan.

Using a CSP approach to tackle time restrictions at execution time is a very promising working line. Experiments show that even using a completely uninformed CSP, this approach brings significant benefits at a very low cost, specially if we consider to obtain the same gains from a planning perspective. This makes us keep on considering a separate process for allocating a plan in time. This process could be used not only to handle delays or temporal user requirements but also all kind of restrictions that come out when a plan is to be instantiated in a particular temporal setting.

References

Biundo, S.; Holzer, R.; and Schattenberg, B. 2004. Project planning under temporal uncertainty. In *ECAI Workshop on Planning and Scheduling: Bridging Theory to Practice*.

Bresina, J.; Dearden, R.; Meuleau, N.; Ramakrishnan, S.; Smith, D.; and Washington, R. 2002. Planning under continuous time and resource uncertainty: A challenge for ai. In *Proc. AIPS-02 Workshop on Planning for Temporal Domains*.

Detcher, R. 2003. *Constraint processing*. Elsevier Science.

Do, M. B., and Kambhampati, S. 2001. Planning as constraint satisfaction: Solving the planning graph by compiling it into csp. *Artificial Intelligence* 132(2):151–182.

Edelkamp, S., and Hoffmann, J. 2004. Pddl 2.2: the language for the classical part of ipc-04. In *ICAPS-2004 - International Planning Competition*, 2–6.

Fox, M., and Long, D. 2003. PDDL 2.1 : An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

Gerevini, A.; Saetti, A.; and Serina, I. 2004. Planning in pddl2.2 domains with lpg-td. In *International Planning Competition, 14th Int. Conference on Automated Planning and Scheduling (ICAPS-04), abstract booklet of the competing planners*.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated planning: Theory and Practice*. Morgan Kauffman.

Kautz, H. 2004. Satplan04: Planning as satisfiability. In *International Planning Competition, 14th Int. Conference on Automated Planning and Scheduling (ICAPS-04), abstract booklet of the competing planners*.

Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *3rd. Int. Conf. on Principles of Knowledge Representation and Reasoning*, 103–114.

Temporal Planning with Preferences and Probabilities

Robert Morris[†], **Paul Morris**[†], **Lina Khatib**[*†]
[*]QSS Group, Inc.
[†]Computational Sciences Division
NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035, USA
{morris,pmorris,lina}@email.arc.nasa.gov

Neil Yorke-Smith
Artificial Intelligence Center
SRI International
Menlo Park, CA 94025, USA
nysmith@ai.sri.com

Abstract

In an uncertain world, a rational planning agent must simultaneously reason with uncertainty about expected outcomes of actions and preferences for those outcomes. This work focuses on systematically exploring the interactions between preferences for the durations of events, and uncertainty, expressed as probability distributions about when certain events will occur. We expand previous work by introducing a means for representing events and durations that are not under the control of the planner, as well as quantitative beliefs about when those events are likely to occur. Two reasoning problems are introduced and methods for solving them proposed. First, given a desired overall preference level, compute the likelihood that a plan exists that meets or exceeds the specified degree of preference. Second, given an initial set of beliefs about durations of events, and preferences for times, infer a revised set of preferences that reflect those beliefs.

Introduction

Rational agents are capable of mentally exploring the interactions between what they believe and what they desire as outcomes of actions. More often than not, the value of the outcomes of actions cannot be described by a single attribute, but rather by attributes that combine to determine the overall value of the outcome (Keeney & Raifa 1993). Furthermore, the outcome of actions may not be known with certainty, as a result of the need to interact with the world.

Many practical planning or scheduling problems surround events that are not controlled by the planning agent. For example, Earth Science observation scheduling may involve assigning times for the remote sensing of an area of interest on the Earth before, during, or after a fire has occurred there. The start and end of the fire are not known with certainty at planning time, but Earth Science models might be available to estimate a set of times when fires are likely to occur. In addition, the scientific utility of an observation may vary based on when the observation is taken relative to the fire, resulting in preferences for temporal orderings and durations between planned events and uncontrollable events (Morris *et al.* 2004b). As automated planning matures as a software technology, new techniques inspired by decision theory are being integrated to address the fact that plans are executed in the world, with varying degrees of value to the planner based on their outcomes (Blythe 1999). A principled approach to scheduling problems as the above is essential for a decision-theoretic temporal planner that takes into account preferences when determining plan quality.

The goal in this paper is to devise systematic methods for exploring the interactions between temporal preferences and uncertainties. We introduce a framework that generalizes the *Simple Temporal Problem* (STP) formulation (Dechter, Meiri, & Pearl 1991), called the *Simple Temporal Problem with Preferences and Probabilities*, or STP^3. One component of the generalization adds the capability to express preferences for times, following (Khatib *et al.* 2001). The other component allows for the designation of uncontrollable events and the associated probability space over times. We extend techniques previously used to solve temporal planning problems with preferences to identify solutions that are both globally preferred and highly probable.

Besides defining the STP^3 framework, the contribution of this paper is to describe solutions to two practical reasoning problems arising from the interactions between probabilities and preferences. We extend techniques previously used to solve temporal problems with preferences to identify solutions that are both globally preferred and highly probable.

Decision-theoretic planning is surveyed by (Blythe 1999). Most approaches extend classical planning techniques or employ Markov Decision Processes (e.g. (Boutilier, Dean, & Hanks 1999)), in contrast to our constraint-based focus. Of work on temporal reasoning for planning, a characteristic example is (Hanks, Madigan, & Gavrin 1995), who, like us, consider exogenous events, but focus on eliciting probabilities and qualitative preferences from a human expert.

In the constraints literature, preferences are commonly represented using semiring-based formulations, the approach we adopt. An alternative formulation for qualitative preferences is CP-nets (Boutilier *et al.* 2004). Uncertainty has also been represented both qualitatively and quantitatively; probabilistic frameworks include that of (Fargier *et al.* 1995), which we adopt, and its extensions.

Generic constraint-based frameworks that account for both preferences and uncertainty include (Dubois, Fargier, & Prade 1996). Our work is distinguished by restricting attention to Simple Temporal constraints. Prior work in this line has considered STPs with preferences but no uncertainty (Khatib *et al.* 2001); and STPs with uncertainty constraints but no preferences (Morris, Muscettola, & Vidal

2001; Tsamardinos, Pollack, & Ramakrishnan 2003). While (Rossi, Venable, & Yorke-Smith 2004) incorporate both aspects, that work considers only qualitative uncertainty, that is, with implied uniform distributions.

Example: Earth Science Campaign Observation Scheduling

An *Earth Science campaign* is a systematic set of activities undertaken to meet a particular science objective. Here, we present a hypothetical campaign based on a science objective to test an emissions model predicting the aerosols released by wildfires. Data on several variables must be gathered in order to accomplish the analysis, and several remote sensors, such as those on the Landsat satellite, provide data products at various spatial resolutions relevant to these variables. Preferred times for acquiring Landsat data for vegetation type for a region of interest in the northern hemisphere would be the prior June or July in the same year that the fire burned, when forested land can most easily be spectrally distinguished from grassland. For mapping aerosol concentration, images coincident to burning must be obtained; the Terra and/or Aqua satellites have relevant instruments. For the burned area, data should be acquired after (though not too long after) the fire is out, while for mapping vegetation moisture content, hyperspectral data from an EO-1 Hyperion instrument are relevant, and the most useful data would be that acquired just preceding the fire.

From this description, the inputs to a campaign planning problem potentially consist of the following characteristics:

- a set of temporal, spatial, and resource constraints on when and where images are to be taken;
- user preferences for when an observation should be taken;
- temporal ordering constraints between planned events and uncontrollable, exogenous events such as fires.

A reasonable goal, given these inputs, is to generate a concise representation of the set of solutions (assignments of times and sensing resources) that are maximally preferred and reflect a set of initial beliefs about when exogenous events are likely to occur. We formulate a framework capable of describing the problem and generating this output.

Simple Temporal Problems with Preferences and Probabilities

A *soft temporal constraint* depicts restrictions on the distance between arbitrary pairs of distinct events, and a user-specified preference for a subset of those distances. In Khatib et al. (Khatib *et al.* 2001), a soft temporal constraint between events i and j is defined as a pair $\langle I, f_{ij}\rangle$, where I is a set of intervals $\{[a,b], a \leq b\}$ and f_{ij} is a *local preference function* from I to a set A of admissible preference values.[1] When I is a single interval, a set of soft constraints defines a *Simple Temporal Problem with Preferences* (STPP), a generalization of a Simple Temporal Problem (Dechter, Meiri, & Pearl 1991). An STPP can be depicted as a pair (V, C) where V is a set of variables representing events or other time-points, and $C = \{\langle [a_{ij}, b_{ij}], f_{ij}\rangle\}$ is a set of soft constraints defined over V. An STPP, like an STP, can be organized as a network of variables representing events, and links labeled with constraint information.

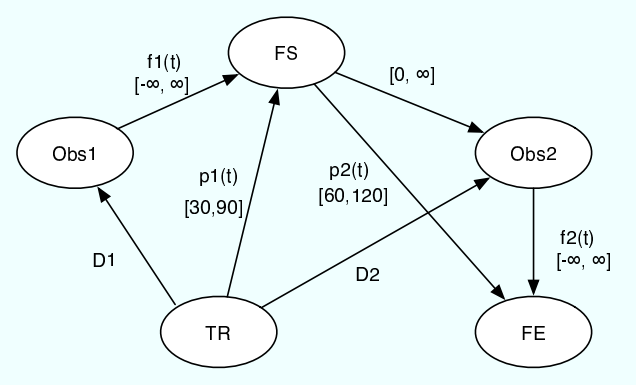


Figure 1: STP[3] Representing the Fire Campaign Scenario

Similar to other recent approaches (Morris, Muscettola, & Vidal 2001; Tsamardinos, Pollack, & Ramakrishnan 2003; Rossi, Venable, & Yorke-Smith 2004), we extend the STPP framework to represent temporal uncertainty. First, we partition V into two groups: the *decision variables* V_d and the *parameters* V_u representing uncontrollable events. This partition induces a distinction between *decision constraints* (C_d) and *uncertainty constraints* (C_u): those constraints whose end-point is controllable (i.e. a decision variable), and those whose end-point is a uncontrollable (i.e. a parameter). An uncertainty constraint depicts a duration between events as a continuous random variable. To ease the exposition, we assume that the uncertainty constraints are mutually independent[2]; this allows the constraints in C_u to be expressed in the form $\langle [a_{ij}, b_{ij}], p_{ij}\rangle$, where $p_{ij} : [a_{ij}, b_{ij}] \rightarrow [0,1]$ is the probability density function over the designated interval. We call the framework $\langle V_d, V_u, C_d, C_u\rangle$, where C_d are soft constraints, a *Simple Temporal Problem with Preferences and Probabilities*, or STP[3].

Example 1 *Earth Science Observation Problem.* Inputs: Variables in V_d standing for two controllable events consisting of taking an observation ($Obs1$, $Obs2$), and two uncontrollable events in V_u, the start and end of a fire (FS, FE) (for simplicity, observations are viewed as instantaneous), as shown in Figure 1. There is also an event TR representing the beginning of time. Soft constraints $f_1(t), f_2(t)$ in C_d are associated with the durations between $Obs1$ and FS, and between $Obs2$ and FE, respectively. For example, $f_1(t)$ may express that there is no value for taking $Obs1$ after the start of the fire (FS), and a preference for times that are as close to FS as possible. Similarly, $f_2(t)$ expresses a preference for $Obs2$ happening before FE as close as possible,

[1] For the purposes of this paper, we assume the values in A are totally ordered, and that A contains designated values for minimum and maximum preference.

[2] For instance, imagine that the Earth Science planner maintains a Bayes network elsewhere to express the dependencies; each probability $p(t)$ is given implicitly by that network.

with a penalty if the observation is taken after the fire. Uncertainty constraints p_1, p_2 in C_u are associated with random variables representing the start time and the duration of the fire. These constraints are based on Earth Science models about fires in the area of interest. For example, p_1 may express a normal distribution over the range of times.

A *solution* to an STP[3] is a set of assignments to $V = V_d \cup V_u$ that satisfies all the constraints in $C = C_d \cup C_u$. Given an STP[3] P, let $Sol(P)$ be the set of all solutions to P. An arbitrary solution $s \in Sol(P)$ can be viewed as having two parts: s_d, the set of values assigned to V_d, and s_u, the set of values assigned to V_u.

Our goal is to develop efficient methods for generating a concise, graphical representation of subsets of $Sol(P)$ corresponding to highly likely, globally preferred solutions. This STP-based graphical representation is called a *flexible (temporal) plan*. Many planning systems use an STP-based representation of the temporal aspects of their plans (Smith, Frank, & Jónsson 2000).

Following previous efforts, methods for flexible temporal planning under uncertainty can be distinguished based on assumptions about the strategy to be applied in executing the flexible plan. A *static execution strategy* assumes no access to the values of s_u during plan execution; by contrast a *dynamic execution strategy* is applied as plan execution proceeds and the values of s_u are observed over time (Morris, Muscettola, & Vidal 2001; Rossi, Venable, & Yorke-Smith 2004). The results of this paper assume a static execution strategy; we defer discussions of planning for dynamic execution of STP[3]s to future work.

Component Solvers. The solution methods described below are based on different decompositions of an STP[3] into component sub-problems for which efficient solution methods exist. As a final preliminary, we fix some terminology and briefly summarize these sub-problem solution methods. Given an STP[3], the *underlying STPP* is the problem that results when a constraint $\{[a, b], p_{XY}\} \in C_u$ is replaced by the STP component constraint $[a, b]$. The *underlying Probabilistic STP* is the problem that results when each soft constraint $\{[a, b], f_{XY}\} \in C_d$ is replaced by the STP component constraint $[a, b]$. The *underlying STP* replaces all constraints in $C_d \cup C_u$ with their STP components.

Efficient solution methods for STPs are well-known (Dechter, Meiri, & Pearl 1991). An Simple Temporal Network (STN) is a graph of nodes representing the STP variables and edges labeled with the interval temporal constraints. Each STN is associated with a distance graph derived from the upper and lower bounds of the interval constraints. An STN is consistent iff the distance graph does not contain a negative cycle; this condition can be determined by applying a single-source shortest path algorithm such as Bellman-Ford. In addition to consistency, it is often useful to determine for an STN the *equivalent* STN (in terms of a set of solutions) in which all the intervals are as "tight" as possible. This *minimal network* can be determined by applying an All-Pairs Shortest Path (APSP) algorithm to the input network (Dechter, Meiri, & Pearl 1991).

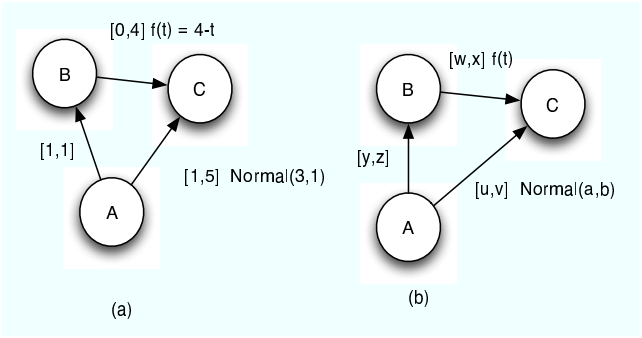


Figure 2: Illustrating the Interactions between Temporal Probabilities and Preferences

Previous efforts in solving STPPs have been based on identifying and applying criteria for "globally preferred solutions" such as "weakest link" (maximize the least preferred local preference), "pareto", and "utilitarian" (Morris *et al.* 2004a). Developing efficient solvers has required local preference functions that are linear or semi-convex.[3] One method for solving STPPs efficiently is called the *chop method*, first introduced in (Khatib *et al.* 2001). The chop method is a two-step search process of iteratively choosing a preference value α, "chopping" every preference function at that point, and then solving an underlying STP defined by the interval of temporal values whose preference values lie above the chop line, i.e. $\{x : f(x) \geq \alpha\}$; henceforth, we refer to this as the *chop interval*. The highest chop point that results in a solvable (i.e. consistent) STP produces a flexible plan whose solutions are exactly the optimal solutions of the original STPP (based on the criteria of weakest link). Binary search can be used to select candidate chop points, making the technique for solving the STPP tractable.

Assessing the Likelihood of Achieving Preferred Plans

This section and the next consider two practical reasoning problems involving the interactions of uncertainty and preferences about time, and demonstrate how under certain assumptions they can be solved efficiently using STP[3]s. The first problem addresses the question: *what are the chances of achieving a certain level of global preference, given my belief about the way the world will behave?* To illustrate, consider the simple STP[3] in Figure 2(a). Here, $V_d = \{A, B\}$ and $V_u = \{C\}$, and there are two decision constraints, between B and C and between A and B. B is tightly constrained to occur exactly one time unit after A. The soft constraint BC prefers durations between B and C to be minimal (higher values more preferred); this is expressed by the preference function $f(t) = 4 - t$. The probability density function for AC is represented by specifying the named function (normal) with mean (3) and standard deviation (1).

[3]A function is *semi-convex* if drawing a horizontal line anywhere in the Cartesian plane of the graph of the function is such that the set of X such that $f(X)$ is not below the line forms an interval. Semi-convexity ensures that there is a single interval above any chop point, and hence that the resulting problem is an STP.

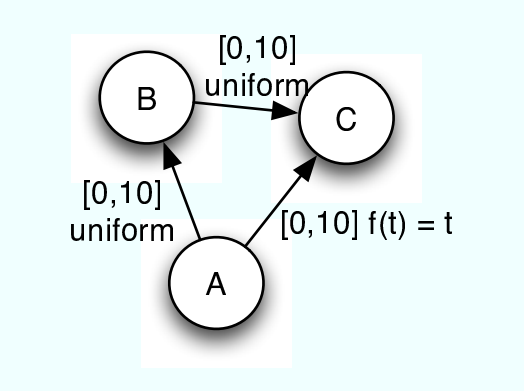


Figure 3: Why the Upper Bound May Not be Tight

Suppose an agent wants to infer the chances of there being a solution with an overall preference level of 2 or greater. We can answer this question by restricting assignments to BC with an f value of 2 or greater, and propagating the temporal constraints over the network. This means shrinking the BC interval to $[0, 2]$, which in turn shrinks AC to $[1, 3]$. Consequently, the answer to the posed question can be obtained by computing $P(1 \leq t \leq 3) = \int_1^3 p(t)dt$.

This technique can be generalized for arbitrary STP^{3}s. Given an STP^3 P, to determine the probability of achieving a solution of global preference value γ or higher, we perform the following procedure:

1. Given an input STP^3, chop each local preference function at the designated preference value γ. Form a new problem by replacing each associated interval with the resulting chop interval.
2. Determine the minimal network of the underlying STP of the new problem, using an APSP algorithm.
3. Compute the overall probability of the underlying probabilistic CSP. Assuming independence of the p_{ij}, the value to be computed is

$$\prod_{p_{ij}} P(a_{ij} \leq t \leq b_{ij}), \quad (1)$$

where for each uncertainty constraint, $[a_{ij}, b_{ij}]$ is the interval of the minimal network derived from step 2.

Provided step 3, which may be done using numerical integration, is of polynomial complexity, the whole method is polynomial. Steps 2 and 3 of this method resemble the method proposed in (Tsamardinos, Pollack, & Ramakrishnan 2003) for solving Probabilistic STPs. Unfortunately, it can be easily shown that the computed value provides only an upper bound on the probability that the solutions defined at that chop level or above will succeed. That this is not a tight upper bound can be demonstrated by a simple example, found in Figure 3. In this example, chopping the preference function at 10 and solving the underlying STP would not shrink the temporal bounds of the uncertainty links. Therefore, the probability of succeeding returned by this method would be 1, although in fact some of the probability mass is lost as a result of the chop.

Despite these limitations, an upper bound computation may be useful; if the bound is too low, the planner will be forced to "lower expectations" of the plan branch under consideration, i.e. its overall expected preference level.

A tighter bound would require examining the mass of the polytope defined by all the constraints (a similar observation was made in (Tsamardinos, Pollack, & Ramakrishnan 2003)). Applied to the previous example, we get $P((0 \leq \text{AB} \leq 10) \wedge (0 \leq \text{BC} \leq 10))$ from (1), but the true probability is $P((0 \leq \text{AB} \leq 10) \wedge (0 \leq \text{BC} \leq 10) \wedge (\text{AB}+\text{BC} \geq 10))$ or simply $P(\text{AB} + \text{BC} \geq 10)$, assuming the bounds. (Note that the AB and BC random variables are no longer independent under the condition $\text{AB} + \text{BC} \geq 10$.) We can reformulate this as $P(\bigvee_x(\text{AB} = x \wedge \text{BC} \geq 10 - x))$ and calculate it as

$$\int_0^{10} \left(\int_{10-x}^{10} p(y)dy \right) p(x)dx.$$

Inducing Preferences from Probabilities

In this section we consider a sort of dual problem to that posed in the previous section: *given current expectations about the world, how can preferences be systematically adjusted to fit with those expectations?* Intuitively, by answering this question, the planner can "factor out" the temporal uncertainty in the problem, resulting in a pure decision problem: because of the factoring, the solutions most preferred based on the induced preferences are also most likely.

This factoring process takes into account the initial preferences on decision constraints, combining them with the preferences induced from the uncertainty constraints. The core idea is to apply the concept of expected utility from decision analysis (Keeney & Raifa 1993) to represent induced local preferences. Once the reasoning is complete, the "output" preferences on the decision constraints thus reflect both the preferences of the agent and its expectation about the uncertainty in the world.

The main result of this section will be to state a set of sufficient conditions for finding an efficient algorithm for this process for certain classes of STP^3. The method consists of:

1. Given an input STP^3, derive the minimal network of the underlying STP.
2. Apply a local consistency algorithm (discussed below) to the resulting STP^3 (i.e. with the tightened interval constraints) to compute the induced preferences.
3. Solve the underlying STPP of the resulting network using the chop solver to find the globally preferred solutions.

The set of solutions making up the flexible plan that results are the *expected globally preferred* solutions.

To examine the second step in more detail, we mimic the method of *triangular reduction* found in (Morris, Muscettola, & Vidal 2001), used to solve Simple Temporal Problems with Uncertainty (STPUs). We consider all STP^{3}s as collections of *triangular subnetworks* of the form illustrated by Figure 2(b), where there is a single uncertainty constraint on AC with bounds $[u, v]$, and two decision constraints on AB and BC with bounds $[y, z]$ and $[w, x]$ respectively. As in the Earth Science example, A might be the beginning of

time, B might be the start of a planned observation, and C the onset of a fire. The goal is to compute the *regression* of p_{AC} over f_{BC} to find the induced soft decision constraint f_{AB}. (The case in which AB is also associated with a soft constraint can be handled as part of the general solution method discussed later.)

To handle the single triangle case, we need to consider three possible orderings between B and C. We assume that step 1 of the approach has been applied, so that the triangular network has been minimized. If B precedes C ($w \geq 0$), then the induced soft constraint is $\{[y, z], f_{AB}\}$, where

$$f_{AB}(t) = \int_u^v f(t' - t)p(t')dt'.$$

Although in general this function cannot be derived analytically, with certain restrictions placed on the shape of the preference function it may be possible to compute it directly. Alternatively, we can estimate it numerically (e.g. Monte Carlo integration), or even perform crude but fast estimation based on the expected value. If C precedes B ($x \leq 0$), then intuitively the planner does not require any knowledge about the expected time of C in order to deduce the preferred time to execute B dynamically (the soft constraint on AB in this case can be derived from that of BC). However, recall that we focus only on the situation of static execution, in which knowledge about C is not available at planning time. This means that the predictive models of the Precede case are relevant to planning the Follow case: the same technique can be followed. Finally, for static execution the same also applies if B and C are unordered ($w < 0, x > 0$).

To derive the induced constraints for general STP^3 networks, we consider all triangles separately, propagating the effects of one operation to neighboring triangles, until the network is quiescent. Thus, the structure of the algorithm is similar to determining path-consistency in an STP network. Propagation requires combining local preference functions. The same combination operator as that used for determining local consistency for preference networks (Rossi *et al.* 2002) can be applied here for propagating soft constraints. After the network has reached quiescence, the planner can safely discard the probability density functions p_{XY} in C_u. Removing them results in the underlying STPP, which can be solved by the chop method (Khatib *et al.* 2001).

The following result summarizes these core ideas. It will be proved informally and illustrated by an example. Following terminology in (Morris, Muscettola, & Vidal 2001), an STP^3 will be said to be *pseudo-controllable* if no interval in an uncertainty constraint is "squeezed" as the result of performing step 1 above (computing the minimal network). We refer to the STP^3 that results from performing step 2 above as the *induced STP^3*.

Theorem 1 Given an STP^3 with the following properties:

1. The input preference functions p_{ij} are linear or semi-convex piecewise linear (intuitively, semi-convex piecewise linear means that there are no "V" shaped segments);
2. The STP^3 is pseudo-controllable;
3. The probability distributions on the uncertainty constraints are normal;

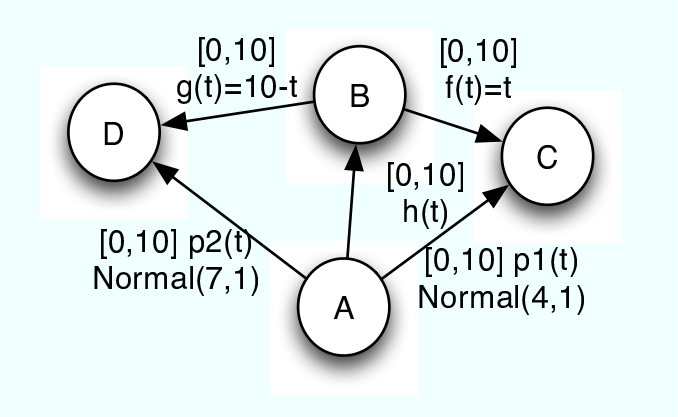


Figure 4: Example of Induced Preferences

then, using the method described above, the set of expected globally preferred solutions to the initial STP^3 can be computed in polynomial time.

The first condition of the theorem is needed to ensure that the induced STP^3 has only functions that are semi-convex, which is required for the application of the chop solver method in step 3 (a polynomial-time procedure). Steps 2 and 3 are required to simplify the induced functions to linear functions involving expected values (see the example below). The conclusion of the proof consists of observing that the underlying procedures applied in the method (all-pairs shortest path, the local-consistency technique for deriving induced preferences, the chop solver, and numerical integration for determining the expected values) are all polynomial.

To illustrate step 2 of the method in the general case, consider the STP^3 in Figure 4. This problem consists of two decision constraints on BC and BD with associated preference functions f, g defined, f clearly preferring larger durations between B and C, and g preferring smaller durations. Two uncertainty constraints on AC and AD consist of normal probability density functions p_1 and p_2 with means and standard deviations indicated in parentheses. The goal is to infer the induced preference function h on AB (the network is already minimal).

First, considering the triangle ABC, one induced function for h arises as follows:

$$\begin{aligned} h_1(t) &= \int_0^{10} f(t' - t)p_1(t')dt' \\ &= \int_0^{10} [t' - t]p_1(t')dt' \\ &= \int_0^{10} t'p_1(t')dt' - t\int_0^{10} p_1(t')dt'. \end{aligned}$$

Notice that because of the pseudo-controllability of the network (it being already minimal), the last equation reduces to $E(T_1) - t$, since then $\int_0^{10} p_1(t')dt' = 1$ and $\int_0^{10} t'p_1(t')dt' = E(T_1)$, where $E(T_1)$ is the expected value of the random variable T_1 associated with the duration. A similar derivation based on the triangle ABD then results in another induced function $h_2(t) = 10 - [E(T_2) - t]$. The

final induced function h becomes the combination of h_1 and h_2: e.g. the intersection of the areas under the functions.

This approach can be generalized for regression over semi-convex piecewise linear preference functions. Let f_{BC} be the intersection of n linear segments $f^1_{BC}, \ldots f^n_{BC}$, where for each k, $[a^k_{BC}, b^k_{BC}]$ is the segment for which $f_{BC} = f^k_{BC}$. When regressing p_{AC} over f_{BC} to compute the induced preference function h_{AB}, we have:

$$h_{AB}(t) = \int_{a^1}^{b^n} f_{BC}(t' - t)p_{AC}(t')dt'$$
$$= \sum_{k=1,\ldots,n} \int_{a^k}^{b^k} f^k_{BC}(t' - t)p_{AC}(t')dt',$$

which simplifies to sums involving linear functions.

This example shows how with suitable restrictions on the shapes of the preference functions and on whether the all-pairs computation eliminates any of the probability mass, the computation of induced preferences can be made efficient.

Discussion and Future Work

We have examined temporal reasoning under the interactions of preferences and quantitative uncertainty in the context of constraint-based planning. In addition to the formulation of the STP^3 framework, which augments the Simple Temporal Problem with both preferences and probabilities, the main contribution of this paper is to formulate two planning decision problems. Utilizing standard methods from decision theory, probability theory, and recent advances in constraint satisfaction, we have shown how flexible temporal plans can be generated that are most preferred based on what the planning agent believes about the expected times of events; and how the agent can update its preferences, given its beliefs.

Fundamentally, preferences and uncertainty are orthogonal aspects of the decision problem. Both planning decisions we have considered are approaches to combining the two aspects; which is most relevant depends on the aim of the planning agent and the questions being asked of it. The first decision, to evaluate the probability of a plan existing with at least a given preference, is useful to determine whether a plan branch can meet a minimum quality threshold. The second decision, to update preferences based on beliefs, is useful to factor the uncertainty into a single criterion for plan evaluation. Besides these two decision problems, the proposed framework can be applied to related problems; for instance, an agent might seek to determine the maximal preference level at which a solution exists with a given probability p. When $p = 1$ and the probabilities are uniform, this corresponds to certain forms of strong controllability addressed in (Rossi, Venable, & Yorke-Smith 2004).

Future theoretical efforts include characterizing more fully the computational complexity of STP^{3}s, and refining the bound on the probability that a plan exists with given preference quality. In addition to implementing the methods described in this paper, our major next step is to extend the results here to address issues in planning under a dynamic execution strategy. Of particular importance will be to examine the interactions between preferences and *wait constraints* that emerge when determining the controllability of flexible plans, as described in (Morris & Muscettola 2000).

Acknowledgments. We thank the reviewers for their comments. The work of the last author on this material is supported by the Defense Advanced Research Projects Agency, through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010.

References

Blythe, J. 1999. Decision-theoretic planning. *AI Magazine* 20(2):37–54.

Boutilier, C.; Brafman, R. I.; Domshlak, C.; Hoos, H. H.; and Poole, D. 2004. Preference-based constrained optimization with CP-Nets. *Computational Intelligence* 20:137–157.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *JAIR* 11:1–94.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–95.

Dubois, D.; Fargier, H.; and Prade, H. 1996. Possibility theory in constraint satisfaction problems: Handling priority, preference and uncertainty. *Applied Intelligence* 6(4):287–309.

Fargier, H.; Lang, J.; Martin-Clouaire, R.; and Schiex:, T. 1995. A constraint satisfaction framework for decision under uncertainty. In *Proc. of UAI 1995*, 167–174.

Hanks, S.; Madigan, D.; and Gavrin, J. 1995. Probabilistic temporal reasoning with endogenous change. In *Proc. of UAI 1995*, 245–254.

Keeney, R., and Raifa, H. 1993. *Decisions with Multiple Objectives*. Cambridge University Press.

Khatib, L.; Morris, P.; Morris, R.; and Rossi, F. 2001. Temporal reasoning about preferences. In *Proc. of IJCAI-01*, 322–327.

Morris, P., and Muscettola, N. 2000. Execution of temporal plans with uncertainty. In *Proc. of AAAI-2000*, 491–496.

Morris, P.; Morris, R.; Khatib, L.; Ramakrishnan, S.; and Bachmann, A. 2004a. Strategies for global optimization of temporal preferences. In *Proc. of CP'04*, 408–422.

Morris, R.; Dungan, J.; Khatib, L.; Bachman, A.; Gasch, J.; Hempel, P.; Williams, J.; Wood, T.; and Bruno, C. 2004b. Coordinated science campaign planning for earth observing missions. In *Proc. of ESTC'04*.

Morris, P.; Muscettola, N.; and Vidal, T. 2001. Dynamic control of plans with temporal uncertainty. In *Proc. of IJCAI-01*, 494–502.

Rossi, F.; Venable, K.; Khatib, L.; Morris, P.; and Morris, R. 2002. Two solvers for tractable temporal constraints with preferences. In *Proc. of Workshop on Preferences in AI and CP*.

Rossi, F.; Venable, K.; and Yorke-Smith, N. 2004. Controllability of soft temporal constraint problems. In *Proc. of CP'04*, 588–603.

Smith, D.; Frank, J.; and Jónsson, A. 2000. Bridging the gap between planning and scheduling. *Knowledge Eng. Review* 15(1).

Tsamardinos, I.; Pollack, M. E.; and Ramakrishnan, S. 2003. Assessing the probability of legal execution of plans with temporal uncertainty. In *Proc. of ICAPS'03 Workshop on Planning Under Uncertainty and Incomplete Information*.

Schedule robustness through *Solve-and-Robustify*: generating flexible schedules from different fixed-time solutions

Nicola Policella, Amedeo Cesta and **Angelo Oddi**
Planning & Scheduling Team
Institute for Cognitive Science and Technology - CNR
Rome, Italy
name.surname@istc.cnr.it

Stephen F. Smith
The Robotics Institute
Carnegie Mellon University
Pittsburgh, USA
sfs@cs.cmu.edu

Abstract

In previous work, we have defined a two-step procedure called *Solve-and-Robustify* for generating flexible, partial order schedules. This partitioned problem solving approach — first find a viable solution and then generalize it to enhance robustness properties — has been shown to provide an effective basis for generating flexible, robust schedules while simultaneously achieving good quality with respect to optimization objectives. This paper extends prior analysis of this paradigm, by investigating the effects of using different start solutions as a baseline to generate partial order schedules. Two approaches are compared: the first constructs partial order schedules from a single fixed-time schedule, obtained by first performing an extended makespan optimization search phase; the second considers the search for fixed-time schedules and flexible schedules in a more integrated fashion, and constructs partial order schedules from a number of different fixed-times starting solutions. The paper experimentally shows how the characteristics of the fixed-time solutions may lower the robustness of the final partial order schedules and discusses the reasons for such behavior.

Introduction

In previous work (Policella *et al.* 2004b; 2004a) we have shown how a two-step procedure — first find a solution then make it robust — can provide an effective basis for generating flexible, robust schedules that also achieve good solution quality. Under this scheme, a feasible fixed-time schedule is generated in stage one (in particular an early start times solution is identified), and then, in the second stage, a procedure referred to as *chaining* is applied to transform this fixed-time schedule into a *Partial Order Schedule*, or $\mathcal{POS}$. By virtue of the fact that a $\mathcal{POS}$ retains temporal flexibility in the start times of various activities in the schedule, a $\mathcal{POS}$ provides greater robustness in the face of executional uncertainty.

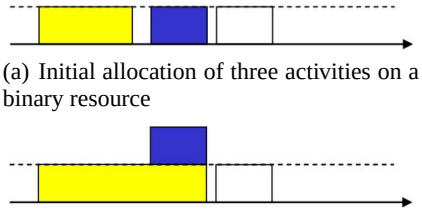
The common thread underlying the "chained" representation of the schedule is the characteristic that activities which require the same resource units are linked via precedence constraints into precedence chains. Given this structure, each constraint becomes more than just a simple precedence. It also represents a *producer-consumer* relation, allowing each activity to *know* the precise set of predecessors which will *supply* the units of resource it requires for execution. In this way, the resulting network of chains can be interpreted as a flow of resource units through the schedule; each time an activity terminates its execution, it passes its resource unit(s) on to its successors. It is clear that this representation is robust if and only if there is temporal slack that allows chained activities to move "back and forth". Concepts similar to chaining have also been used elsewhere: for example, the Transportation Network introduced in (Artigues & Roubellat 2000), and the Resource Flow Network described in (Leus & Herroelen 2004) are based on equivalent structural assumptions.

This paper addresses an aspect not explored in our previous work: how different start schedules influence the whole process of identifying partial order schedules. Taking a constraint-based solver and the best chaining algorithm from our previous work, we define and analyze the performance of two extended search configurations of the *Solve-and-Robustify* solution approach: the first combination schema constructs flexible schedules from a single starting point solution after an extended, iterative sampling optimization phase, while the second iterates both solve and robustify considering different fixed-time schedules as starting points and selecting the best partial order schedule found.

The paper first introduces the basic concepts of schedule robustness and Partial Order Schedules, then describes the two step approach to $\mathcal{POS}$ synthesis. The new analysis follows next. We describe each extended search procedure, present the results of an experimental evaluation and interpret these results. Finally we draw some conclusions.

Scheduling with Uncertainty and Partial Order Schedules

The usefulness of schedules in most practical scheduling domains is limited by their brittleness. Though a schedule offers the potential for a more optimized execution than would otherwise be obtained, it must in fact be executed as planned to achieve this potential. In practice this is generally made difficult by a dynamic execution environment, where unforeseen events quickly invalidate the schedule's predictive assumptions and bring into question the continuing validity of the schedules's prescribed actions. The lifetime of a schedule tends to be very short, and hence its optimizing advan-



(a) Initial allocation of three activities on a binary resource

(b) Allocation after an activity lasts longer

Figure 1: Brittleness of a fixed-time schedule

tages are generally not realized. For instance, let us consider the example in Fig. 1 that shows the allocation of three different activities on a binary resource. According to quite common practice in scheduling, a solution associates an exact start and end time to each activity.

Such solution may exhibit a high degree of brittleness, for instance, as shown in Fig. 1(b), when the first activity lasts longer than expected a conflict in the usage of the machine is immediately generated because of the fixed start-time for the activities.

An alternative approach consists of adopting a graph formulation of the scheduling problem, wherein activities competing for the same resources are simply ordered to establish resource feasibility, and it is possible to produce schedules that retain temporal flexibility where allowed by the problem constraints. In essence, such a "flexible schedule" encapsulates a set of possible fixed-time schedules, and hence is equipped to accommodate some amount of the uncertainty at execution time.

Following this intuition we have introduced the definition of *Partial Order Schedules*, or $\mathcal{POS}$s (Policella 2005). A $\mathcal{POS}$ consists of a set of feasible solutions for the scheduling problem that can be represented in a compact way by a temporal graph, that is, a graph in which any activity is associated to a node and temporal constraints define the order in which such activities have to be executed.

To provide a more formal definition of a $\mathcal{POS}$ we use the activity on the node representation: given a problem P, this can be represented by a graph $G_P(V_P, E_P)$, where the set of nodes $V_P = V \cup \{a_0, a_{n+1}\}$ consists of the set of activities specified in P and two dummy activities representing the origin (a_0) and the horizon (a_{n+1}) of the schedule, and the set of edges E_P contains P's temporal constraints between pairs of activities. A solution of the scheduling problem can be represented as an extension of G_P, where a set E_R of simple precedence constraints, $a_i \prec a_j$, is added to remove all the possible resource conflicts. Given these concepts, a *Partial Order Schedule* is defined as follows:

Definition 1 (Partial Order Schedule) *Given a scheduling problem P and the associated graph $G_P(V_P, E_P)$ that represents P , a* Partial Order Schedule*, $\mathcal{POS}$, is a set of solutions that can be represented by a graph $G_{POS}(V_P, E_P \cup E_R)$.*

In practice a $\mathcal{POS}$ is a set of partially ordered activities such that any possible complete activity allocation that is consistent with the initial partial order is also a resource and time feasible schedule.

It is worth noting that a partial order schedule provides an immediate opportunity to reactively respond to some of the possible external changes by simply propagating their effects over the "graph", by using a polynomial time computation. In fact the augmented duration of an activity, as well as a greater release time, can be modeled as a new temporal constraint to post on the graph. It is also important to note that, even though the propagation process does not consider the consistency with respect the resource constraints, it is guaranteed to obtain a feasible solution by definition of $\mathcal{POS}$s. Therefore a partial order schedule provides a mean to find a new solution and ensures its fast computation.

RCPSP/max. This work considers the Resource-Constrained Project Scheduling Problem with minimum and maximum time lags, RCPSP/max (Bartusch, Mohring, & Radermacher 1988) as the reference problem. The basic entities of this problem are a set of *activities* denoted by $V = \{a_1, a_2, \ldots a_n\}$. Each activity has a fixed *processing time*, or *duration*, p_i and must be scheduled without preemption.

A *schedule* is an assignment of start times to activities $a_1, a_2, \ldots a_n$, i.e. a vector $S = (s_1, s_2, \ldots, s_n)$ where s_i denotes the start time of activity a_i. The time at which activity a_i has been completely processed is called its *completion time* and is denoted by e_i. Since we assume that processing times are deterministic and preemption is not permitted, completion times are determined by $e_i = s_i + p_i$. Schedules are subject to both *temporal* and *resource constraints*. In their most general form temporal constraints designate arbitrary minimum and maximum time lags between the start times of any two activities, $l_{ij}^{min} \leq s_j - s_i \leq l_{ij}^{max}$ where l_{ij}^{min} and l_{ij}^{max} are the minimum and maximum time lag of activity a_j relative to a_i. A schedule $S = (s_1, s_2, \ldots, s_n)$ is *time feasible*, if all inequalities given by the activity precedences/time lags and durations hold for start times s_i. During their processing, activities require specific resource units from a set $R = \{r_1, r_2, \ldots r_m\}$ of resources. Resources are *reusable*, i.e. they are released when no longer required by an activity and are then available for use by another activity. Each activity a_i requires of the use of req_{ik} units of the resource r_k during its processing time p_i. Each resource r_k has a limited capacity of c_k units. A schedule is *resource feasible* if at each time t the demand for each resource $r_k \in R$ does not exceed its capacity c_k, i.e. $\sum_{s_i \leq t < e_i} req_{ik} \leq c_k$. A schedule S is called *feasible* if it is both time and resource feasible.

Metrics for Comparing Partial Order Schedules. As described before, a single $\mathcal{POS}$ represents a set of temporal solutions that are also resource feasible. This set of schedules provides a means for tolerating some amount of execution uncertainty. When an unexpected event occurs (e.g., a start time delay), the temporal propagation mechanism (a polynomial time calculation) can be applied to update the

start times of all activities and, if at least one temporal solution remains viable, produces a new partial order schedule. Therefore, it is intuitive that the quality of a certain $\mathcal{POS}$ is tightly related to the set of solutions that it can represent. In fact the greater the number of solutions, the greater is the expected ability in dealing with scheduling uncertainty. Another aspect to consider in analyzing the solutions clustered into a partial order schedule is the distribution of such alternatives over all the activities. Such distribution will be the result of the configuration given by the constraints present in the solution. For this reason it is necessary to introduce metrics that consider such aspects.

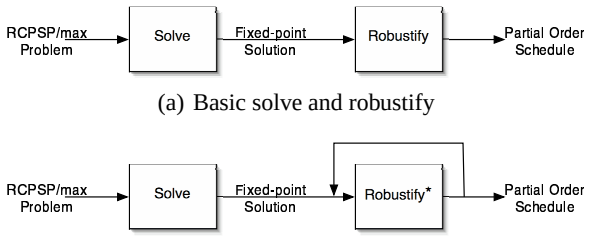
A first measure, $flex$, is taken from (Aloulou & Portmann 2003). This measure counts the *number of pairs of activities in the solution which are not reciprocally related by simple precedence constraints*. This provides a first analysis of the configuration of the solution. The rationale is that when two activities are not related it is possible to move one without moving the other one. Hence, the higher the value of $flex$ the lower the degree of interaction among the activities. It is worth noting that a limitation of the $flex$ metric consists in being able to give only a qualitative evaluation of the solution because it counts only existence/not-existence of a direct ordering relation. This may be sufficient for scheduling problem with no time lag constraints like the one used in (Aloulou & Portmann 2003), but is rather limited to describe flexibility in RCPSP/max where it is necessary to integrate this flexibility measure with some factor that takes into account the quantitative aspects of the temporal problem (or solution).

A metric that satisfies this requirement is defined in (Cesta, Oddi, & Smith 1998). It requires the presence of a fixed time horizon for the termination of all the activities. In order to compare two or more $\mathcal{POS}$s we bound any partial order schedule to have a finite number of solutions. Then the metric is defined as the average width, relative to the temporal horizon, of the temporal slack associated with each pair of activities (a_i, a_j):

$$fldt = \sum_{i=1}^{n} \sum_{j=1 \wedge j \neq i}^{n} \frac{slack(a_i, a_j)}{H \times n \times (n-1)} \times 100 \qquad (1)$$

where H is the temporal horizon of the problem, n is the number of activities, $slack(a_i, a_j)$ is the width of the allowed distance interval between the end time of activity a_i and the start time of activity a_j, and 100 is a scaling factor[1]. We use this metric to characterize the *fluidity* of a solution, i.e., the ability to use flexibility to absorb a temporal variation in the execution of activities. It also considers that a temporal variation concerning an activity is absorbed by the temporal flexibility of the solution instead of generating deleterious domino effect (the higher the value of $fldt$, the less the risk, i.e., the higher the probability of localized changes).

[1] In (Cesta, Oddi, & Smith 1998) this metric was defined as robustness of a solution, RB.



(a) Basic solve and robustify

(b) Improved chaining

Figure 2: Previous work

Solve and Robustify

In (Cesta, Oddi, & Smith 1998) a two-stage approach to generating a flexible schedules is introduced as one possibility for robust schedules. Under this scheme a feasible fixed-time schedule is first generated in stage one, and then, in the second stage, a procedure referred to as *chaining* is applied to obtain a robust solution. In this second step, fixed-time commitments are converted into a sequences (chains) of activities to be executed by various resources.

In a recent paper, (Policella *et al.* 2004b), this approach has been generalized to RCPSP/max. These results establish the basic viability of a chaining procedure. At the same time, the procedure used in this work was developed simply to provide a means of transforming a given schedule into a $\mathcal{POS}$; no attention was given to the potential influence of the chaining procedure itself on the properties exhibited by the final, flexible solution. For these reasons, in (Policella *et al.* 2004a) the authors examine the problem of generating $\mathcal{POS}$s from the broader perspective of producing flexible schedules with good robustness properties, and investigate the design of informed chaining procedure that exploits knowledge of these properties to increase the robustness of the final $\mathcal{POS}$. This latter work first introduced an iterative improvement schema for a randomized chaining algorithm.

A sketchy representation of the *Solve-and-Robustify* approach can be seen, Fig. 2(b), as an open loop cascade of a "solver" and a "robustify" modules. The subsequent work in (Policella *et al.* 2004a) can be represented as a first loop with respect to the simple cascade by applying an iterative improvement cycle around the robusty phase, Fig. 2(b). Iterative improvement is obtained by randomizing the basic procedure to obtain different search paths from different restarts[2].

The ESTA greedy solver. As a solver we use here the *Precedence Constraint Posting* greedy schema called ESTA described in (Cesta, Oddi, & Smith 1998; 2002). This proceeds analyzing the infinite capacity representation of a RCPSP/max problem where the temporal constraint are satisfied and the resource profiles contain violations. More precisely, a violation or *resource contention peak* consists in a set of activities that are executed at the same time and that

[2] In the figure, a module containing a randomized procedure is labeled with a star (e.g., *Robustify**).

require an amount of resource greater than the resource capacity. Then the solving process posts further precedence constraints to remove these peaks. The selection of the new constraint is accomplished by using three alternative heuristics; a first, simple strategy consists in considering all the pairs of activities in a peak. Other two approaches use the MCS, minimal conflict sets, where an MCS is a set of resource conflicting activities such that any of its proper subset is consistent. To avoid the complexity of computing all the MCSs two polynomial sampling approaches are used: *linear* and *quadratic*. The use of the three different heuristics gives rise to three different variants of the ESTA algorithm.

Iterative chaining. In our current approach the robustification phase is carried out through chaining. A chaining procedure transforms a feasible fixed-time solution into a $\mathcal{POS}$ by dispatching activities to specific resource units[3]. Since choices can be made as to how to dispatch activities to resource units, it is possible to generate different $\mathcal{POS}$s from the same starting solution, and these different $\mathcal{POS}$s can be expected to have different robustness properties.

In (Policella *et al.* 2004a) iterative sampling is used to explore this set of solutions. Randomization is added to obtain a different solution at each iteration and, in so doing, to generate a sequence of $\mathcal{POS}$s starting from the same initial schedule. The $\mathcal{POS}$s are evaluated with respect to a specific robustness measure, and the best one found is returned. Additionally, different heuristics have been explored that change the effectiveness of chaining. In this paper we use the most effective one, called MINID (*minimizing interdependencies*) in (Policella 2005).

The heuristic MINID takes into account existing ordering relations with those activities already allocated in the chaining process in order to minimize possible links among pairs of chains which will degrade the flexibility of the solution. In this procedure, the allocation of an activity a_i on the chains of a multi-capacitive resource r_j proceeds according to the following steps:

(1) collect in the set P_{a_i}, the chains k belonging to r_j, for which their last element, $last_k$, is already ordered with respect to the activity a_i, i.e., $last_k \prec a_i$;

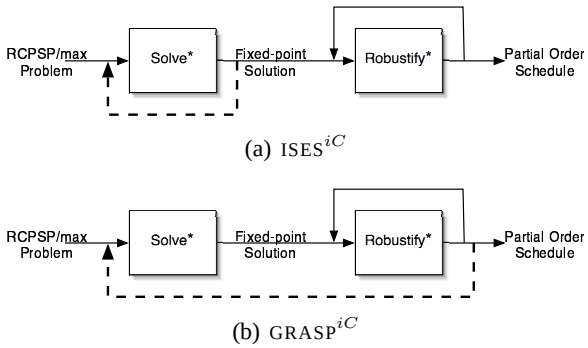
(2) if $P_{a_i} \neq \emptyset$ a chain $k \in P_{a_i}$ is randomly chosen, otherwise a chain k is randomly selected among the available ones;

(3) a constraint $a_k \prec a_i$ is posted, where $a_k = last_k$;

(4) if a_i requires more than one resource unit, then the remaining set of available chains is split into two subsets: the set of chains which has a_k as last element, C_{a_k}, and the set of chains which does not, $\bar{C_{a_k}}$;

(5) to satisfy all remaining resource requirements, a_i is allocated first to chains belonging to the first subset, $k' \in C_{a_k}$ and,

[3]Note that this procedure is required to enable schedule execution.



(a) ISES^{iC}

(b) GRASP^{iC}

Figure 3: Current contribution

(6) in case this set is not sufficient, the remaining units of a_i are then randomly allocated to the first available chains, k'', of the second subset, $k'' \in \bar{C_{a_k}}$.

Some properties of chaining. Why are we interested in the use of the chaining approach? Previous works have found several interesting properties.

First, any partial order schedule can be obtained by a chaining procedure (Policella *et al.* 2004a, Theorem 1). This result allows us to restrict the attention, without loss of generality, to the set of $\mathcal{POS}$s generated by a chaining procedure.

Second, given a schedule S, the makespan of the earliest solution[4] of the partial order schedule generated through chaining, $\mathcal{POS}_S$, is not greater than the makespan of the input solution. Thus, the makespan of a solution is preserved by the chaining-based robustification phase.

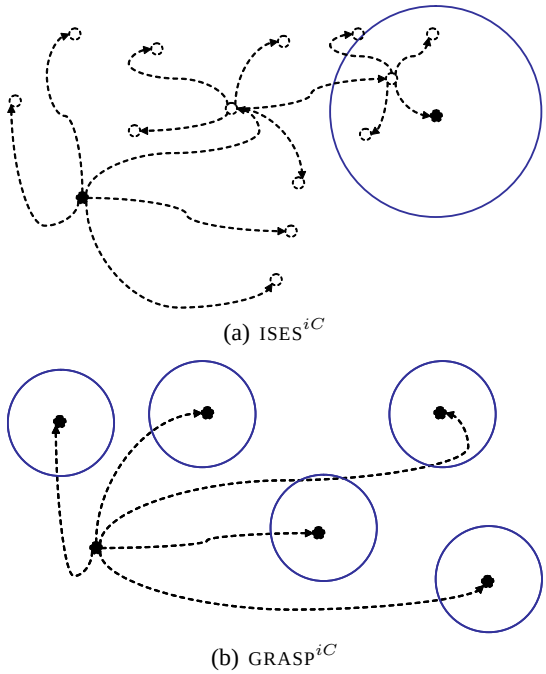
Additionally, previous work has identified structural properties for the effectiveness of chaining in RCPSP/max. In particular, the analysis of solutions obtained via chaining has brought out the presence of *synchronization points* which tend to degrade solution flexibility. These stem from the presence of activities which require multiple resource units and from precedence constraints between activities allocated on different chains. In fact, each of these aspects will mutually constrain two (or more), otherwise independent processes.

Finally, an interesting aspect of the procedure is that it can be coupled with any fixed-time schedule generation procedure. Independently on the fact that we are using our own constraint-based approach to solving RCPSP/max, the robustify step can be applied to other approaches to solve the same scheduling problem like (Dorndorf, Pesch, & Phan-Huy 2000; Smith & Pyle 2004; Cicirello & Smith 2004).

Generating flexible schedules from different initial solutions

In previous work we have not investigated the effects that different fixed-time initial solutions may have to the final

[4]The solution in which each activity is allocated at the earliest start time defined by the partial order schedule.



(a) ISES^{iC}

(b) GRASP^{iC}

Figure 4: ISES^{iC} vs. GRASP^{iC}

synthesized $\mathcal{POS}$. This complementary analysis is contributed by this paper. In particular we extend our schema as shown in Fig. 3 where dashed lines highlight the new steps: (a) understanding the influence on $\mathcal{POS}$s of a much broader search for a better makespan before the robustify step; (b) considering the whole solve-and-robustify open loop within a meta-heuristic schema. For the solving phase we draw on our own CSP solvers for RCPSP/max.

The iterative sampling procedure

The first approach we investigate in shown in Fig. 3(a). Instead of the simple ESTA greedy solver, we insert the ISES iterative improvement schema that is based on a randomized ESTA (hence *Solve** in the figure). The randomized version of ESTA is obtained by doing a pseudo-random selection of the decision. Conflict selection is performed not picking the best ranked conflict but choosing randomly within an acceptance band. This technique, introduced first in (Oddi & Smith 1997), eliminates heuristic bias when the discrimination power of the estimator is low. The whole approach is called ISES^{iC} because in practice it joins the ISES optimizer with the iterative chaining that uses the MINID method.

The grasp-like procedure

The second approach introduced in this paper is shown in Fig. 3(b). It follows a typical GRASP meta-heuristic schema (Resende & Ribeiro 2002). The idea is to use the randomized ESTA to create different start schedule with the greedy solver then apply the iterative chaining exploration (hence the GRASP^{iC} name). This whole schema is iterated until a termination criterion is met. Based on a robustness metric the best $\mathcal{POS}$ found is returned.

A further remark

In Fig. 4 we underscore graphically the different ways the iterative procedure and the GRASP-like follow through the search space. They both start from an infinite capacity solution that is not resource consistent, the ISES^{iC} performs iterative sampling for a while then pick the best fixed-time schedule, according to the makespan value, and call the robustify (sketched as a circle whose size is related to the number of performed iterations — see later). The GRASP^{iC} follows the different pattern of simply looking for different start schedule using the randomized ESTA then calling the robustification.

Experimental Evaluation

We apply the two approaches to two well-known RCPSP/max benchmark sets, j30 and j100. The benchmark set j30 is composed of 270 instances each of them with 30 activities and 5 resources, while the j100 is composed of 540 instances each of them with 100 activities and 4 resources.

Regarding ISES^{iC}, at the first step the ISES algorithm is used considering 10 restarts as termination criterion. After that, a $\mathcal{POS}$ is searched for applying iterative chaining with a number of iterations equals to 100. The implementation of the GRASP^{iC} method is based on 10 iterations of the main loop, where an initial fixed-time schedule is computed at each step, and on 10 iterations in iterative improvement applied to robustify, where a different $\mathcal{POS}$ is computed at each step. Thus in both approaches the best $\mathcal{POS}$ is selected among a set of 100 alternatives. We recall that in both cases the iterative chaining method is used with the MINID heuristic. In order to have a fair evaluation of the two criteria, we normalize the results according to an upper bound. The latter is obtained for each metric $\mu()$ considering the value $\mu(\mathcal{P})$ that is the quality of the network that represents the initial temporal structure of the problem.

Therefore, in the tables below, for each algorithm we report the following values: the normalized values, $|flex|$ and $|fldt|$, of the two robustness metrics, the CPU time requested (in seconds), the number of posted constraints, npc, and the makespan, mk.

The set j30. Table 1 contains the results obtained using the six versions of the ISES^{iC} method[5]. An analysis of the result presented shows us as the values of both the metrics do not show great difference among the different algorithm variants. In fact the values of $|flex|$ range from 0.472 to 0.475 and the results of $|fldt|$ from 0.626 to 0.631. Regarding the two metrics introduce above, it is possible to notice a twofold behavior. In fact the values of $flex$ are close or equals to the best value, 0.475, obtained in (Policella *et al.* 2004a)[6]. This is because in both the procedure the iterative chaining method with MINID has an important rule in increasing the flexibility (in terms of $flex$) of the final partial order

[5]These are obtained by the combination of the three heuristic with the two robustness criteria used during the iterative chaining process.

[6]This result is obtained by using ESTA^{iC}+MINID$_{flex}$ +MCS quadratic with 100 iterations.

j30	$\|$**flex**$\|$	$\|$**fldt**$\|$	**cpu**	**npc**	**mk**
ISES^{iC}+MINID$_{flex}$	0.472	0.603	6.63	30.20	98.52
ISES^{iC}+MINID$_{flex}$ +MCS linear	0.472	0.607	8.22	30.26	98.12
ISES^{iC}+MINID$_{flex}$ +MCS quadratic	0.475	0.601	9.44	30.09	98.26
ISES^{iC}+MINID$_{fldt}$	0.451	0.627	6.63	30.64	98.44
ISES^{iC}+MINID$_{fldt}$ +MCS linear	0.446	0.631	8.22	30.64	98.10
ISES^{iC}+MINID$_{fldt}$ +MCS quadratic	0.448	0.626	9.42	30.74	98.22

Table 1: ISES^{iC}

j30	$\|$**flex**$\|$	$\|$**fldt**$\|$	**cpu**	**npc**	**mk**
GRASP^{iC}+MINID$_{flex}$	0.461	0.668	6.43	29.17	105.25
GRASP^{iC}+MINID$_{flex}$ +MCS linear	0.473	0.672	7.49	28.54	104.65
GRASP^{iC}+MINID$_{flex}$ +MCS quadratic	0.475	0.670	8.54	28.85	104.86
GRASP^{iC}+MINID$_{fldt}$	0.443	0.686	6.43	29.36	105.08
GRASP^{iC}+MINID$_{fldt}$ +MCS linear	0.448	0.687	7.49	28.85	104.56
GRASP^{iC}+MINID$_{fldt}$ +MCS quadratic	0.449	0.689	8.53	28.99	104.59

Table 2: GRASP^{iC}

schedule[7]. On the contrary, the same behavior is not confirmed by the results obtained through the ISES^{iC} variants which try to optimize the $fldt$ metric. In this case we have a decreasing of the quality: from 0.670 to 0.631. A different behavior is obtained in case of the GRASP^{iC} variants (see Table 2). Both considering $flex$ and $fldt$, the methods are able to achieve good quality solutions. In fact in the case of $flex$, like in the case of ISES^{iC}, we achieved results that are close or equals to the current best (0.475). It is worth noting that in the case of $fldt$ the same results obtained in (Policella *et al.* 2004a) are achieved. This confirm the relevance of the improvements in the chaining algorithms.

The set j100. Table 3 and Table 4 show the results obtained in the case of the benchmark j100. In this case the difference of the two problem are smoother than in the previous benchmark. In case of $flex$ we have that the ISES^{iC} variants are better of the GRASP^{iC} (but with an improvement of only 2%). The opposite result is achieved in case of $fldt$: in this case we have 0.642 and 0.634 respectively for the best GRASP^{iC} variant and the best ISES^{iC} variant (the percentage difference is about 1%). The difference behavior observed in case of j100 with respect to the benchmark j30 underscores a distinction between the two benchmarks. In fact, despite the size of each instance, the benchmark j100 presents a set of instances that are simpler with respect to the j30, both in terms of the temporal network and in the resource usage.

Discussion: makespan versus robustness

The tradeoff between makespan and robustness highlighted above is worth a comment. We have seen before the results of the ISES^{iC} method which is based on the use of ISES to optimize the makespan of the initial schedule as well as to increase the efficiency of the solving process. These results show that optimization of the makespan value reduces the ability to obtain robust schedules especially if the fluidity metric is taken into account.

Figure 5 underlines possible motivations which can lead to different behaviors between flexibility and fluidity with respect to the makespan optimization factor. Fig. 5(a) presents a possible problem. This is a one resource problem in which three activities have to be scheduled. These are ordered according to the constraints in the figure, i.e., between a and b there is a simple precedence constraint[8] while between a and c the constraint specifies a time window $[1, 3]$, i.e., c cannot start more than 3 time-units after or 1 time-unit before the end of the activity a. Furthermore, the size of each activity describes both its duration (the width) and its resource need (the height). Additionally both a and b require one resource unit for two time units while c has a duration equal to three time-units and a resource requirement of 2. To complete the description of the problem, the resource capacity is equal to 2.

Figure 5(b) and 5(c) show two different fixed-time schedules with the associated partial order schedule (the darker arrows represent the additional constraints necessary to obtain a flexible solution). Note that in this case for each fixed-time solution there is a unique partial order schedule: in fact any of the two proposed solutions gives a complete linearization of the activities. The two schedules have different makespan, respectively 7 and 8.

Let us now consider first the flexibility metric. If we look at the two partial order schedules (on the right hand side of Fig. 5(b) and 5(c)) it is possible to notice that in both cases there is the same $flex$ value. In fact in both cases the $flex$ value is zero because there is no pair of un-ordered ac-

[7]Of course there is a noticeable difference in terms of CPU time, where we have respectively 9.44 seconds for the ISES^{iC}+MINID$_{flex}$ +MCS quadratic version and 1.21 for the ESTA^{iC}+MINID.

[8]Where H represents the temporal horizon of the problem.

j100	**|flex|**	**|fldt|**	**cpu**	**npc**	**mk**
ISES^{iC}+MINID$_{flex}$	0.211	0.620	44.80	42.57	414.73
ISES^{iC}+MINID$_{flex}$ +MCS linear	0.211	0.619	43.63	42.22	413.67
ISES^{iC}+MINID$_{flex}$ +MCS quadratic	0.211	0.617	46.05	42.20	413.69
ISES^{iC}+MINID$_{fldt}$	0.205	0.634	44.85	42.37	414.61
ISES^{iC}+MINID$_{fldt}$ +MCS linear	0.205	0.633	43.64	42.11	413.40
ISES^{iC}+MINID$_{fldt}$ +MCS quadratic	0.205	0.632	46.00	42.17	413.47

Table 3: ISES^{iC}

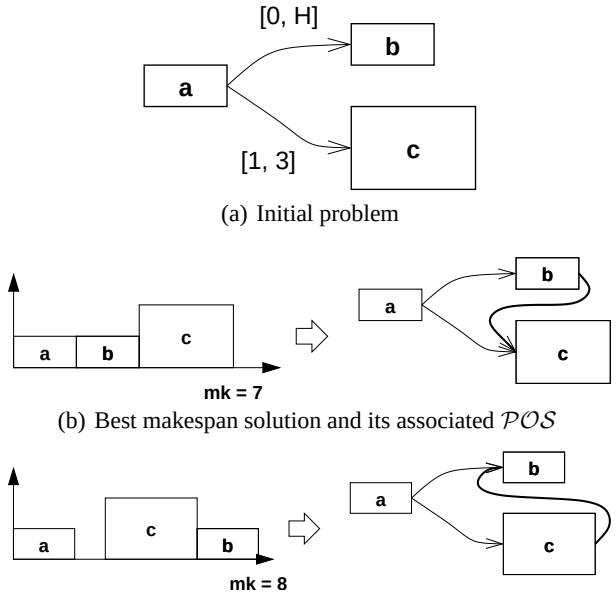
j100	**|flex|**	**|fldt|**	**cpu**	**npc**	**mk**
GRASP^{iC}+MINID$_{flex}$	0.207	0.630	26.67	38.80	437.77
GRASP^{iC}+MINID$_{flex}$ +MCS linear	0.207	0.627	27.01	38.94	437.20
GRASP^{iC}+MINID$_{flex}$ +MCS quadratic	0.208	0.629	27.85	38.84	437.46
GRASP^{iC}+MINID$_{fldt}$	0.201	0.641	26.74	39.11	436.32
GRASP^{iC}+MINID$_{fldt}$ +MCS linear	0.201	0.641	27.01	38.79	435.72
GRASP^{iC}+MINID$_{fldt}$ +MCS quadratic	0.202	0.642	27.85	38.77	435.79

Table 4: GRASP^{iC}

tivities. Now we can shift the attention toward the fluidity metric. This metric considers the slack value between any pair of activities, that is, the minimum and maximum distance between them. In the figure, for the pair (a, c) we have the same value for both solutions, $dist(a, c) = [1, 3]$, which stems from the time window constraint defined between the two activities. On the other pair instead we have two different values: $dist(a, b) = [0, 1]$ in the case in Fig. 5(b) and $dist(a, b) = [4, H - 4]$ in the case in Fig. 5(c). This clearly shows that the flexibility value for the sub-optimal solution is greater than the one in Fig. 5(b). The same behavior can be found for the pair (b, c), where we have $dist(b, c) = [0, 1]$ for the case in Fig. 5(b) while for the case in Fig. 5(c) $dist(c, b) = [0, H - 8]$. The problem is that in the schedule with the optimum makespan (Fig. 5(b)) the activity b is "caged in" by the other two activities. Therefore the time window constraint defined between a and c has the effect of limiting the flexibility of activity b. Furthermore since the capacity of the resource is equal to the requirement of c, no chaining method can overcome this problem.

It is possible to note that this is peculiar characteristic of the RCPSP/max problems and in particular this is due to the presence of maximum distance constraints. In fact if the maximum constraint between a and c did not exist, activity b would have the ability to move back and forth in a larger interval thus yielding in a more flexible solution.

Further remarks. The use of more optimized initial schedule biases the robustify phase against the construction of flexible partial order schedule. In fact the tightness (makespan) of the initial solution can preclude the achievement of good solutions. This behavior is justified from the different nature of the two metrics: $flex$ is a qualitative criterion whereas the $fldt$ is a more quantitative metric. Therefore a schedule with a better makespan value presents an allocation of the activities more compact giving fewer degree of intervention to the iterative chaining procedure. Finally



(a) Initial problem

(b) Best makespan solution and its associated $\mathcal{POS}$

(c) A sub-optimal solution and its associated $\mathcal{POS}$

Figure 5: Dependency between makespan and robustness

while in (Policella *et al.* 2004a) has been proved that the robustify step does not deteriorate the results obtained in the solve phase, we can not claim the same for the effects of some characteristics of the fixed-time solutions on the final flexible schedule.

Conclusions

This work has considered the Solve-and-Robustify approach introduced in (Policella *et al.* 2004b; 2004a) and has complemented previous results by analyzing the influence of dif-

ferent fixed-time schedule on the final partial order schedule. In particular, two more sophisticated approaches based on Solve-and-Robustify have been defined and evaluated. The former is based on the idea of developing flexible solutions from makespan optimized solutions. The second approach instead has been obtained following the GRASP paradigm, where the robustify step can be considered as a local search procedure.

The results of the experimental evaluation have shown an interesting trade-off between the makespan and the more quantitative of the robustness metrics used, $fldt$. In practice the use of more optimized initial schedule biases the robustify phase against the construction of flexible partial order schedule. In fact the tightness (makespan) of the initial solution can preclude the achievement of good flexible solutions. Therefore while the robustify step does not deteriorate the results obtained in the solve phase some characteristics of the fixed-time solutions may lower the robustness of the final partial order schedules.

Acknowledgments

Amedeo Cesta, Angelo Oddi, and Nicola Policella's work is partially supported by MIUR (Italian Ministry for Education and Research) under project RoboCare. Stephen F. Smith's work is supported in part by the U.S. Air Force Research Laboratory - Rome, under contract F30602-02-2-0149, by the National Science Foundation under contract # 9900298 and by the CMU Robotics Institute.

References

Aloulou, M. A., and Portmann, M. C. 2003. An Efficient Proactive Reactive Scheduling Approach to Hedge against Shop Floor Disturbances. In *Proceedings of* 1^{st} *Multidisciplinary International Conference on Scheduling : Theory and Applications (MISTA 2003)*, 337–362.

Artigues, C., and Roubellat, F. 2000. A polynomial activity insertion algorithm in a multi-resource schedule with cumulative constraints and multiple modes. *European Journal of Operational Research* 127(2):297–316.

Bartusch, M.; Mohring, R. H.; and Radermacher, F. J. 1988. Scheduling project networks with resource constraints and time windows. *Annals of Operations Research* 16:201–240.

Cesta, A.; Oddi, A.; and Smith, S. F. 1998. Profile Based Algorithms to Solve Multiple Capacitated Metric Scheduling Problems. In *Proceedings of the* 4^{th} *International Conference on Artificial Intelligence Planning Systems, AIPS-98*, 214–223.

Cesta, A.; Oddi, A.; and Smith, S. F. 2002. A Constraint-based method for Project Scheduling with Time Windows. *Journal of Heuristics* 8(1):109–136.

Cicirello, V. A., and Smith, S. F. 2004. Heuristic selection for stochastic search optimization: Modeling solution quality by extreme value theory. In *Principles and Practice of Constraint Programming,* 10^{th} *International Conference, CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, 197–211. Springer.

Dorndorf, U.; Pesch, E.; and Phan-Huy, T. 2000. A time-oriented branch-and-bound algorithm for resource-constrained project scheduling with generalised precedence constraints. *Management Science* 46:1365–1384.

Leus, R., and Herroelen, W. 2004. Stability and Resource Allocation in Project Planning. *IIE Transactions* 36(7):667–682.

Oddi, A., and Smith, S. F. 1997. Stochastic Procedures for Generating Feasible Schedules. In *Proceedings* 14^{th} *National Conference on Artificial Intelligence (AAAI-97)*, 308–314.

Policella, N.; Oddi, A.; Smith, S. F.; and Cesta, A. 2004a. Generating Robust Partial Order Schedules. In *Principles and Practice of Constraint Programming,* 10^{th} *International Conference, CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, 496–511. Springer.

Policella, N.; Smith, S. F.; Cesta, A.; and Oddi, A. 2004b. Generating Robust Schedules through Temporal Flexibility. In *Proceedings of the* 14^{th} *International Conference on Automated Planning & Scheduling, ICAPS'04*, 209–218. AAAI.

Policella, N. 2005. *Scheduling with Uncertainty: a Proactive Approach using Partial Order Schedules*. Ph.D. Dissertation, University of Rome "La Sapienza".

Resende, M., and Ribeiro, C. 2002. Greedy Randomized Adaptive Search Procedures. In Glover, F., and Kochenberger, G., eds., *Handbook of Metaheuristics*. Kluwer Academic Publishers. 219–249.

Smith, T. B., and Pyle, J. M. 2004. An effective algorithm for project scheduling with arbitrary temporal constraints. In *Proceedings of the* 19^{th} *National Conference on Artificial Intelligence, AAAI-04*, 544–549.

Stratified Heuristic POCL Temporal Planning based on Planning Graphs and Constraint Programming

Ioannis Refanidis

University of Macedonia, Dept. of Applied Informatics, Thessaloniki, Greece
yrefanid@uom.gr

Abstract

This paper elaborates on the temporal planning graphs with mutual exclusion reasoning of the known TGP planning system, in order to build a heuristic temporal Partial Order Causal Link (POCL) planner. The planner exploits the temporal planning graph in two ways: First, it obtains heuristic estimates for sets of open goals during the plan construction phase, in order to solve the symbolic dimension of the planning problem. And second, it obtains additional disjunctive constraints, based on both permanent and temporary mutex relations, which are used by a CSP solver, in order to solve the temporal dimension of the planning problem. Furthermore, in the paper we simplify the temporal planning graph construction process, we propose two completeness preserving pruning rules and a heuristic function that takes into account clusters of permanently mutexed open goals. Preliminary results evaluate the effectiveness of the various choices and indicate future research directions.

Introduction

Partial Order Causal Link (POCL) planning was the dominant planning paradigm until the middle of the previous decade. It has been adopted by numerous planning systems, such as SNLP (McAllester and Rosenblitt, 1991) and UCPOP (Penberthey and Weld, 1992) for symbolic domains, (Penberthy and Weld, 1994) for temporal and metric domains and (Peot and Smith, 1992; Pryor and Collins, 1996) for domains with uncertainty, among others. The main problem of the POCL approach was its inability to solve problems of moderate size, although it provably managed to reduce the size of the search space, with respect to the older state-space or regression-based planning approaches. This was mainly due to the lack of effective domain independent heuristics. Researchers mainly concentrated on devising alternative flaw selection strategies, e.g. (Peot and Smith, 1993; Joslin and Pollack, 1994; Schubert and Gerevini, 1995) among others.

The introduction of planning graphs (Blum and Furst, 1997) caused a significant increase in the performance of planning systems. State-space planners (Bonet, Loerings and Geffner, 1997; Refanidis and Vlahavas, 2001; Bonet and Geffner, 2001; Hoffmann and Noebel, 2001) exploited planning graph structures to devise heuristics to guide either progression or regression, achieving great performance, both in terms of solution speed and in the size of the tractable problem instances. TGP (Smith and Weld, 1999) extended planning graphs in a temporal setting.

The first attempt to exploit planning graph structures for POCL planning in symbolic domains was RePOP (Nguyen and Kambhampati, 2001). (Younes and Simmons, 2003) utilize also planning graphs to extract heuristics in a POCL setting, but they emphasize in flaw selection strategies.

Constraint programming techniques have been used in POCL, especially in temporal domains, before the advent of planning graphs and graph-based heuristics. Examples are IxTeT (Ghallab and Laruelle, 1994), ZENO (Penberthy and Weld, 1994) and Deviser (Vere 1983). These techniques are very efficient in reasoning about the time-windows. A common approach adopted by the POCL planning community is to interleave planning and scheduling, solving conflicts by introducing simple ordering constraints between actions (Smith, Frank and Jonsson, 2000; Vidal and Geffner, 2004).

In this paper we utilize the temporal planning graphs with mutual exclusion reasoning (Smith and Weld, 1999) to create a heuristic POCL planner that will use constraint programming to schedule the actions. Our temporal planning setting consists of deadline goals being simultaneously achieved at the end of the plan execution, whereas durative actions achieve their effects at the end of their duration. Our approach consists of two phases: Initially a temporal planning graph, involving eternal and conditional mutex relations between pairs of propositions, pairs of actions and pairs of a proposition and an action, is created until leveling off. Then, a POCL planning phase is started, progressively solving open goals and posting disjunctive constraints to solve threats. Heuristics extracted from the temporal planning graph are used to select among the alternative ways to support open goals. Threats are detected using the information encoded in mutex relations. Our approach does not rely on the presence of no-op actions or of a minimum quantum of time. After a plan with no open goal is found, a CSP solver is employed to search for a feasible schedule. Suitable completeness preserving pruning rules have been employed to reduce the branching factor.

According to (Smith, Frank and Jonsson, 2000), our approach could be classified as stratified P&S. However, there is some interleaving between P&S, since non-

disjunctive temporal constraints, originating by the causal link relations, are taken into account during the plan construction phase and may lead to plan rejection, in case of tight bounds. Moreover, temporal information is taken into account by the heuristic.

The rest of the paper is as follows: First we review the temporal planning graph structure, as it has been implemented in our system. Then we rewrite the graphplan plan-extraction phase, using POCL planning and constraint programming. On top of this, we build then a heuristic POCL temporal planner, with the heuristic being extracted from the planning graph. In the same section we present two additional pruning techniques that we found useful in our experiments. Finally, we present preliminary results from the evaluation of the various techniques implemented in our system and pose future directions.

Temporal Planning Graphs

In this section we present the way the temporal planning graphs are constructed in our system. We use a slightly different and simplified approach than in (Smith and Weld, 1999), which does not lead to arbitrarily complex formulae.

As in (Smith and Weld, 1999), we define eternal mutex (emutex) relations and conditional mutex (cmutex) relations between pairs of propositions, pairs of actions and pairs of a proposition and an action (in the following we will refer to propositions and actions as nodes of the planning graph). An emutex relation between two nodes denotes that they can never hold simultaneously (being held for an action means being executed). A cmutex relation between two nodes determines a time point, before which the two nodes cannot hold simultaneously (this time point may be infinite, denoted with *inf*). In the following we will use $Prec(A)$ and $Eff(A)$ to denote the preconditions and effects of an action A respectively.

Two nodes are emutexed in the following cases:

- Each proposition P is emutexed with its negation $\neg P$.
- An action A and a proposition P are emutexed, iff $P \in Eff(A)$ or $\neg P \in Eff(A)$ or $\neg P \in Prec(A)$.
- Two actions A and B are emutexed iff either A or B deletes the preconditions or effects of the other, or A and B have emutexed preconditions.

Cmutex relations are computed in a recursive manner, during the construction of the planning graph. The basic data structures of the temporal planning graph are the following:

- $prop(P,T)$: Time T is the earliest time proposition P may become true.
- $action(A,T)$: Time T is the earliest time action A may start execution.
- $emutex(N_1,N_2)$: Nodes N_1 and N_2 are emutexed.
- $cmutex(N_1,N_2,T)$: Nodes N_1 and N_2 are cmutexed until time T (T may be *inf*).

To avoid duplicate effort in storing mutex relations, we used a lexicographic ordering in their arguments N_1 and N_2.

During planning graph creation, a time-ordered queue of events is maintained. There are two kind of events:

- $new\_prop(P,T)$: Proposition P has been achieved at time T.
- $end\_cmutex(P,Q,T)$: The cmutex relation between propositions P and Q ends at time T.

The procedure for creating the temporal planning graph is initialized by asserting $prop(P,0)$ and $new\_prop(P,0)$ for each initial state's proposition P. Then, the planning graph is created according to the following steps (Figure 1):

1. Retract the earliest event E. Let T be its time. If no event exists, then stop.
2. If $E=new\_prop(P,T)$, let *Actions* include each action A having P in its preconditions. If $E=end\_cmutex(P,Q,T)$ event, let *Actions* include each action A having P and Q in its preconditions.
3. For each action A in *Actions*, such that all of its preconditions have been achieved in time earlier than or equal to T and there is no pair of preconditions being emutexed or cmutexed until later than T:
   a. Call ADD_EFFECTS($Eff(A)$, $T+D_A$)
   b. Call NEW_EMUTEX(A)
   c. Call CMUTEX_ACTION_PROP1(A,T)
   d. Call CMUTEX_ACTIONS(A,T)
   e. Call CMUTEX_PROPS(A)
   f. Call STOP_CMUTEX_PROPS(A,$T+D_A$)
   where D_A is A's duration.

In the following we explicate these procedure calls.

ADD_EFFECTS($Eff(A)$,$T+D_A$): For each firstly achieved fact $P \in Eff(A)$, this procedure asserts $prop(P,T+D_A)$ and event $new\_prop(P,T+D_A)$. Moreover, *emutex* relations to existing nodes are created. For those propositions in $Eff(A)$ that already exist in the planning graph with time reference earlier than or equal to $T+D_A$, nothing happens. Finally, for those propositions in $Eff(A)$ that already exist in the planning graph, with time reference later than $T+D_A$, their time reference is replaced by $T+D_A$ and the corresponding pending *new_prop* event is updated accordingly.

NEW_EMUTEX(A): This procedure creates *emutex* relations between the new action A and existing planning graph nodes, according to the related definitions.

CMUTEX_ACTION_PROP1(A,T): This procedure creates new *cmutex* relations between the newly inserted action A and already existing propositions that are cmutexed with A's preconditions and are not emutexed with A. For each such proposition Q that is cmutexed with any proposition $P \in Prec(A)$ until time T_1, a cmutex relation between A and Q until (the maximum possible) T_1 is asserted.

CMUTEX_ACTIONS(A,T): This procedure creates cmutex relations between A and other existing actions, to which A is not emutexed. A is cmutexed to another action B until time T_1, if there is a pair of propositions $P \in Prec(A)$ and $Q \in Prec(B)$, such that $cmutex(P,Q,T_1)$, and T_1 is the latest time for any such pair of propositions, provided that $T_1 > T$.

CMUTEX_PROPS(A): This procedure creates cmutex relations between propositions achieved by A, and other

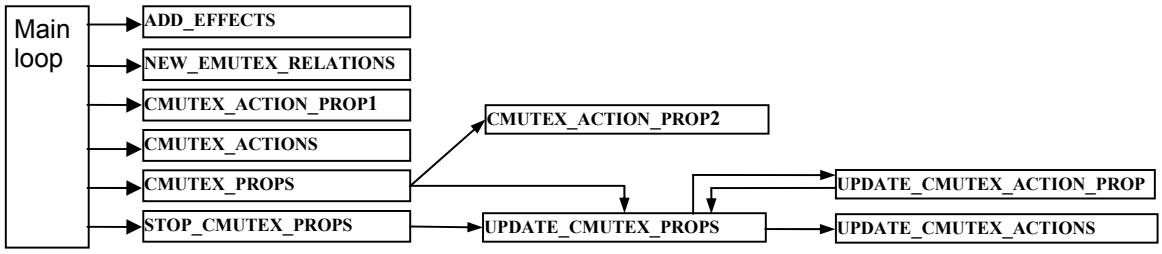


Figure 1: Flow of procedure calls while creating temporal planning graphs.

propositions. In particular:

- Each proposition P firstly achieved by A is cmutexed until *inf* to every other proposition Q, to which A is emutexed and P is not emutexed. A call to CMUTEX_ACTION_PROP2(A,P,Q,*inf*) is issued.
- Each proposition P firstly achieved by A is cmutexed to every other proposition Q, to which A is cmutexed. The end T_1 of *cmutex*(P,Q,T_1) is equal to the sum between T_0 and D_A, where T_0 is obtained by *cmutex*(A,Q,T_0). Event *end_cmutex*(P,Q,T_1) is created, in case T_1 is not *inf* (T_1 is *inf* if T_0 is also *inf*). A call to CMUTEX_ACTION_PROP2(A,P,Q,T_1) is issued.
- For each proposition P reachieved by A, for every other proposition Q to which A is cmutexed until time $T_0<inf$, call UPDATE_CMUTEX_PROPS(P,Q,T_1), where $T_1=T_0+D_A$.

STOP_CMUTEX_PROPS(A,$T+D_A$): This procedure terminates cmutex relations between propositions in the following four cases:

- For each pair of propositions P and Q, such that $P\in Eff(A)$, $Q\in Eff(A)$, call UPDATE_CMUTEX_PROPS(P,Q,$T+D_A$).
- For each proposition $P\in Eff(A)$, for each proposition $Q\in Prec(A)$, such that $\neg Q\notin Eff(A)$, call UPDATE_CMUTEX_PROPS(P,Q,$T+D_A$).
- For each proposition $P\in Eff(A)$, for each proposition $Q\notin Prec(A)$, such that $\nexists\ R\in Prec(A)$, such that *emutex*(Q,R) or *cmutex*(Q,R,T_1), for some T_1, call UPDATE_CMUTEX_PROPS(P,Q,$T+D_A$).
- For each newly achieved proposition $P\in Eff(A)$, for each action $B\neq A$, such that P is not mutexed with B, for each proposition $Q\in Eff(B)$, call UPDATE_CMUTEX_PROPS(P,Q,$T+D_A+D_B$)

The following procedures were referenced above:

CMUTEX_ACTION_PROP2(A,P,Q,T): This procedure creates cmutex relations until T between proposition P and actions that have Q as precondition, i.e. for each action $B\neq A$, such that $Q\in Prec(B)$, assert *cmutex*(P,B,T).

UPDATE_CMUTEX_PROPS(P,Q,$T+D_A$): This procedure updates *cmutex* relations between existing propositions (note that the absence of a *cmutex* relation between a pair of existing propositions means that they are not cmutexed). So, if *cmutex*(P,Q,T_1), where $T_1>T+D_A$, exists in the planning graph, then retract *cmutex*(P,Q,T_1), delete the event *end_cmutex*(P,Q,T_1) (if any), assert *cmutex*(P,Q,$T+D_A$), create event *end_cmutex*(P,A,$T+D_A$) and perform the following calls:

- UPDATE_CMUTEX_ACTION_PROP(P,Q,$T+D_A$)
- UPDATE_CMUTEX_ACTIONS(P,Q,$T+D_A$)

Finally, the following two procedures are responsible for recursively updating cmutex relations concerning actions:

UPDATE_CMUTEX_ACTION_PROP(P,Q,$T+D_A$): This procedure updates any preexisting cmutex relation between proposition P and any action C, such that $Q\in Prec(C)$, as well as between proposition Q and any action B, such that $P\in Prec(B)$. Concerning the former case (the second is symmetric), the update is performed only if, taken into account the rest of C's preconditions and the cmutex relations between them and P, the resulting end time T_1 of the cmutex relation between P and C is earlier than the existing one. For each such update, and for each $R\in Eff(C)$, UPDATE_CMUTEX_PROPS(P,R,T_1+D_C) is called, where D_C is the duration of C.

UPDATE_CMUTEX_ACTIONS(P,Q,$T+D_A$): This procedure updates any preexisting cmutex relation between pairs of actions B and C, such that $P\in Prec(B)$ and $Q\in Prec(C)$. The update is performed only if, taken into account the cmutex relations between any pair of preconditions of B and C, the resulting end time T_1 of the cmutex relation between B and C is earlier than the existing one.

Efficient planning graph construction

With a closer look at the presented algorithm we consider that cmutex relations between actions are not used for updating other types of cmutexes; they are just updated when changes in cmutex relations between propositions or pairs of an action and a proposition occur. So, we could omit the calls to **CMUTEX_ACTIONS** and **UPDATE_CMUTEX_ACTIONS** from the presented algorithm and replace them by a separate procedure that would be called once only, at the end of the algorithm, and would compute the cmutex relations between actions based on the cmutex relations between propositions.

Note that computing and updating cmutex relations between actions is the most costly part of the planning graph construction phase. This observation led us to a more greedy approach: We do not compute cmutex relations between actions in advance, but we compute them on demand during the plan extraction phase. In case of a heuristic plan extraction phase this leads to a significant reduction in the time needed to solve the problems, since the majority of the cmutex relations between actions is never requested.

Plan Extraction as a Constraint Satisfaction Problem

In this section we formulate the plan extraction process as a constraint satisfaction problem in the space of partial plans. Our approach works with ground actions and represents their start times with constraint variables.
The main data structures are the following:

- An *Agenda* with entries of the form $\langle G,T\rangle$, where G is an open goal and T is a constraint variable, always unified with the start time of an action of the plan.
- A partial plan *Plan* with entries of the form $\langle A,T_A\rangle$, denoting that action A starts execution at time T_A, where T_A is a constraint variable.
- A list *Persistence* with entries of the form $\langle G,T_1,T_2\rangle$, meaning that proposition G has to be true between T_1 and T_2, where T_1 and T_2 are constraint variables.

Agenda is initialized with the goal propositions and the time at which the goals have firstly achieved in a non-mutexed way, whereas the other structures are initially empty. The main loop of the algorithm is the following:

1. Dequeue $\langle G,T\rangle$ from *Agenda*. If *Agenda* is empty, call the CSP solver to schedule the actions. In case of success, return current instantiated plan, otherwise backtrack.
2. Non-deterministically try to support $\langle G,T\rangle$ in one of the following options:

2a. Support $\langle G,T\rangle$ from the initial state. Call:
PERSISTENCE_PLAN(G,0,T,*Plan*)
PERSISTENCE2(G,0,T,*Persistence*)
Add $\langle G,0,T\rangle$ in *Persistence*.

2b. Non-deterministically try to support $\langle G,T\rangle$ from an existing action $\langle A,T_A\rangle$, such that $G\in Eff(A)$. Post the constraint $T_A+D_A\le T$. Call:
PERSISTENCE_PLAN(G,T_A+D_A,T,*Plan*)
PERSISTENCE2(G,T_A+D_A,T,*Persistence*)
Add $\langle G,T_A+D_A,T\rangle$ in *Persistence*.

2c. Non-deterministically choose a new action A to insert in *Plan*, such that $G\in Eff(A)$. Create a new constraint variable, T_A, for the start time of A. Post the constraint $T_A+D_A\le T$. Call:
ACTION_PLAN(A,T_A,*Plan*)
ACTION_PERSISTENCE(A,T_A,*Persistence*)
PERSISTENCE_PLAN(G,T_A+D_A,T,*Plan*)
PERSISTENCE2(G,T_A+D_A,T,*Persistence*)
Add $\langle G,T_A+D_A,T\rangle$ in *Persistence*. For each precondition P of A, add $\langle P,T_A\rangle$ in *Agenda*.

In the following we explicate the procedure calls.

PERSISTENCE_PLAN(G,T_1,T_2,*Plan*): This procedure ensures that protecting goal G between T_1 and T_2 is consistent with the current plan. In particular, for every entry $\langle A,T_A\rangle$ in *Plan*:

– If there is an *emutex* relation between A and G, or a *cmutex* that lasts until *inf*, post the following constraint:
$T_A\ge T_2$ or $T_A+D_A\le T_1$
– If there is a *cmutex* relation between A and G that ends at finite time T, post the following constraint:
$T_A+D_A\le T_1$ or $T_A\ge T_2$ or $T_1\ge T$ or $T_A\ge T$

PERSISTENCE2(G,T_{11},T_{12},Persistence): This procedure ensures that protecting goal G between T_{11} and T_{12} is consistent with other protected intervals. In particular, for every entry $\langle G_2,T_{21},T_{22}\rangle$ in *Persistence*:

– If G_1 and G_2 are emutexed or cmutexed until *inf*, post the following constraint:
$T_{11}\ge T_{22}$ or $T_{21}\ge T_{12}$
– If G_1 and G_2 are cmutexed until the finite time T, post the following constraint:
$T_{11}\ge T_{22}$ or $T_{21}\ge T_{12}$ or $T_{11}\ge T$ or $T_{21}\ge T$

ACTION_PLAN(A,T_A,*Plan*): This procedure resolves threats between the new action A and other existing actions in *Plan*. In particular, for every entry $\langle B,T_B\rangle$ in *Plan*:

– If $A=B$ (different instances of the same action), or if A and B are emutexed, or if they are cmutexed until *inf*, post the following constraint:
$T_B\ge T_A+D_A$ or $T_A\ge T_B+D_B$
– If A and B are cmutexed until the finite time T, post the following constraint:
$T_B\ge T_A+D_A$ or $T_A\ge T_B+D_B$ or $T_B\ge T$ or $T_A\ge T$

ACTION_PERSISTENCE(A,T_A,*Persistence*): This procedure resolves threats between the new action A and the protected intervals in *Persistence*. In particular, for every entry $\langle G,T_1,T_2\rangle$ in *Persistence*:

– If A and G are emutexed or cmutexed until *inf*, post the following constraint:
$T_A+D_A\le T_1$ or $T_A\ge T_2$
– If A and G are cmutexed until the finite time T, post the following constraint:
$T_A+D_A\le T_1$ or $T_A\ge T_2$ or $T_A\ge T$ or $T_1\ge T$

The innovation in the above algorithm is that temporary *cmutex* relations are taken into account and produce threats, whereas alternative ways to promotion, and demotion are available to resolve these threats. An interesting remark concerns the way these threats are resolved. The four disjuncts in e.g.:

$T_{11}\ge T_{22}$ or $T_{21}\ge T_{12}$ or $T_{11}\ge T$ or $T_{21}\ge T$

do not constitute mutual exclusive cases (as it happens with the promotion and demotion in solving threats caused by permanent mutexes). Actually in a solution plan it may be the case that more than one of these disjuncts is true. An alternative settlement that produces mutual exclusive cases would be the following one:

($T_{11}\ge T_{22}$ and $T_{11}<T$ and $T_{21}<T$) or
($T_{21}\ge T_{12}$ and $T_{11}<T$ and $T_{21}<T$) or
($T_{11}\ge T$ and $T_{21}<T$) or
$T_{21}\ge T$

This resolution poses more constraints and may lead faster to fail, but it also produces plans with constraints that are not intuitive.

Heuristic POCL Temporal Planning

As with the original TGP, the main problem of the plan extraction process described in the previous section is that it must be repeated for each time step, starting from the time point at which the goals are firstly achieved without mutexes between them, where the time step is equal to the greatest common divisor of the durations of the problem actions. This is necessary if we want to get an optimal plan with respect to its makespan. However, this is unpractical for realistic domains, especially for temporal ones, so we disregard optimality and we attempt to obtain near optimal plans, using heuristics extracted by a leveled-off planning graph and branch and bound search techniques.

There are several ways to derive heuristics from planning graphs, which can easily be adapted in a temporal setting. The most used is the sum heuristic (Nguyen and Kambhampati, 2000), which scores a partial plan with the sum of the earliest possible times, in which each proposition G in *Agenda* can be achieved. These times are obtained by the $prop(G,T)$ relations of the temporal planning graph.

In our system we adopted a slightly different heuristic, adapted by (Younes and Simmons, 2003). For each set of open goals:

- We do not consider duplicate goals, i.e. multiple occurrences of the same proposition in *Agenda*. Each multiple occurring proposition is counted once only.
- We do not consider goals that can potentially be supported by actions already inserted in the plan. This favors actions that, although they inserted in the plan to support a specific open goal, could be used to support other goals too.
- From the remaining goals, we sum the maximum of the heuristic values for each cluster of goals that are emutexed or cmutexed until the infinite to each other. This takes into account that permanently mutexed propositions have usually to be achieved in sequence, so the cost to achieve them is estimated by the cost of the most expensive one.
- In the above sum, we add the number of the goals, taking into account both duplicate goals, goals supported by existing actions and mutexed goals. This aims at solving ties between partial plans.

The procedure for POCL heuristic planning is quite similar to that described in the previous section. The basic difference is the following: For a specific partial plan, a number of refined new plans are created simultaneously, corresponding to the various ways to support an open goal. The new plans are scored by the heuristic function and stored in memory together with other existing partial plans. That is, we adopted a best-first schema to search the space of the partial plans. Finally, the best partial plan is retracted and refined in the next step. Ties are broken by preferring newer plans.

This setting led us in a variety of choices that reduce the branching factor without sacrificing completeness. The first one of them concerns the delay of the ordering decisions: threats in the plan are resolved by posting disjunctive constraints, instead of creating separate child-plans without disjunctions. Having disjunctions reduces the possibilities for propagating constraints, which may lead to inability for early inconsistency detection. On the other hand, this choice overcomes the problem of the numerous similar child-plans having the same sets of open goals and actions but different orderings. Generating separate child plans for the alternative ways to resolve threats would lead to a continuously increasing number of similar plans with equal heuristic values that would dominate every other area of the search space. In case of an early wrong estimate, it would be more difficult to overcome it.

Another tuning of the planning process concerns the way open goals are selected. We adopted the approach to resolve the most costly goal of each selected partial plan, according to the heuristic function. Note that the order in which open goals are selected does not constitute a backtracking point, however, it has large impact on the efficiency of any POCL planner. Selecting the most costly goal allows generally for faster downgrading of the heuristic values, towards a solution plan. On the other hand, with our heuristic function, which is a hybrid max-sum function, selecting any other than the most costly goal could lead to the generation of child-plans with the same heuristic value to their parent.

Repeated Subgoal Pruning

In this section we present a pruning technique we implemented in our planner, which in some problems led to significant reduction in the branching factor. We call it *repeated subgoal pruning* and, in order to present it we have first to define the notion of primitive subgoal chains.

Def. 1: A *primitive subgoal chain* is an ordered list of subgoals $\langle G_n, G_{n-1}, \ldots G_0 \rangle$, where each subgoal G_i has been inserted in *Agenda* as a precondition of action A_i, where action A_i was initially inserted in the plan to support subgoal G_{i-1}. Subgoal G_0 is an original goal of the problem instance. We denote the primitive subgoal chain starting by any specific subgoal G with PCHAIN(G).

Note that in Def. 1 we do not care about other subgoals that action A_i might have been used to support, after its introduction in the plan. So, the preconditions of A_i cannot belong to the same primitive subgoal chain with these subgoals that A_i was used afterwards to support. A primitive subgoal chain is extended only when a new action is introduced in the plan to support its left-most subgoal. In this case, various new primitive subgoal chains are generated, one for each precondition of the new action.

Repeated subgoal pruning rule: A new action A with $G \in Eff(A)$, cannot be inserted in a plan to support a specific subgoal G, if there is any proposition $P \in Precs(A)$ such that $P \in$ PCHAIN(G).

The above pruning rule sacrifices completeness. Indeed, there are many plans where a single proposition may change its truth value many times. For example, in Figure 2 the task is to clean a dirty room, which can be done only if the light is *on*. The light is initially *off*, and we have to leave it *off* at the end. So, in order to achieve all goals, light has to get once *on* and then *off* in the plan. On the other hand, changing repeatedly and many times the status of the light would be irrational. We want to avoid irrational loops without sacrificing completeness. To achieve this, we adopted the following approach:

The repeated subgoal pruning rule is implemented in our system as follows: Suppose we are processing subgoal $\langle G,T\rangle$. There are several possibilities to support G (from the initial state, from several existing actions and from several new actions) and all of them are examined in order to generate the corresponding child-plans. Suppose there are some (among others) new actions that could support G, but the pruning rule is activated for all of them. For all these actions, a single new child-plan is generated, in which no new action has been inserted. The new child-plan differs from its parent one only in that subgoal $\langle G,T\rangle$ has been marked as inactive (initially all subgoals are marked as active). Moreover, $\langle G,T\rangle$ has been labeled with the number of actions the current plan has. In case no active goal exists in the new *Agenda*, the new child plan is pruned.

Inactive subgoals differ from active ones in that they cannot be supported by new actions or by the initial state: They can only be supported by existing actions that have been added in the plan after they became inactive (this is why each inactive subgoal is labeled with the number of actions in the plan at the time the subgoal became inactive).

The child-plan generated when a subgoal becomes inactive is identical to its parent one, so it is scored with the same value by the heuristic function. So, it may happen that this child-plan will be selected in the next loop of the POCL planning procedure for further refinement, thus entering in an infinite loop. We cope with this problem in two ways: First, we penalize inactive subgoals, so that their plans will not be selected in case other plans with the same heuristic value but fewer inactive subgoals exist. This is done by adding for each inactive subgoal a small constant in the heuristic value. Second, when a plan with inactive subgoals is selected for further refinement, the inactive subgoal is not selected as the most costly subgoal (event if it is), except if new actions have been inserted into the plan after this subgoal became inactive. This last technique eliminates the risk of entering into infinite loops when refining partial plans with inactive subgoals.

Figure 2 illustrates all these. We have two subgoals, *off* and *clean*. Suppose that initially we choose SWITCH_OFF to support *off*, because this is the most costly subgoal. So, the primitive subgoal chain $\langle on, off\rangle$ is generated from the precondition of SWITCH_OFF. Suppose next we have to support *clean*, and we insert CLEAN in the plan. The primitive subgoal chains $\langle on,clean\rangle$ and $\langle dirty,clean\rangle$ are generated. Suppose next we have to support the

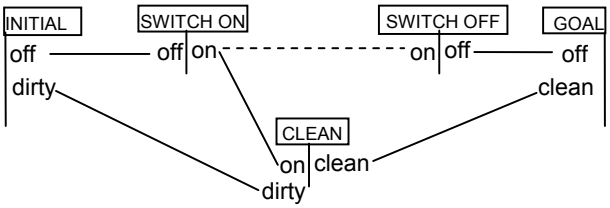


Figure 2: Subgoal chains. Solid lines denote primitive subgoal chains.

precondition *on* of SWITCH_OFF. The only way to do this is to insert SWITCH_ON in the plan. But this activates the pruning rule, and the insertion is rejected. The subgoal becomes inactive and then subgoal *on* of CLEAN is tried. Now, action SWITCH_ON can be inserted into the plan, since subgoal chain $\langle off,on,clean\rangle$ does not activate the pruning rule. Next, subgoal *dirty* is supported by the initial state, subgoal *on* of SWITCH_OFF can be supported by the already existing action SWITCH_ON (which has been inserted into the plan after the subgoal became inactive) and finally subgoal *off* is supported by the initial state.

An alternative, but less efficient way to cope with repeated subgoals would be to prune altogether any child-plan with new actions falling under the pruning rule, without marking goals in *Agenda* as inactive. In this case, in order to preserve completeness, all subgoals in *Agenda* should be processed for each partial plan (and not only the most costly one). This approach, although more general and intuitive, is less efficient.

The last paragraph gives rise to an alternative formulation of the pruning rule. First, we generalize the notion of the primitive subgoal chains:

Def. 2: A *subgoal chain* is an ordered list of subgoals $\langle G_n, G_{n-1}, \ldots, G_0\rangle$, where each subgoal G_i is a precondition of some action A of the plan, where an effect of A supports subgoal G_{i-1}. Subgoal G_0 is an original goal of the problem instance.

This definition does not care about the order in which actions have been inserted in the plan and subgoals were resolved, as happens with primitive subgoal chains. With this definition, the pruning rule can be rewritten in a more declarative (but less operational) way:

> For any subgoal G appearing in a causal link of a partial order plan, there must exist at least one subgoal chain starting with G, were G is not repeated.

One could claim that this pruning rule is nothing more than repeated state pruning, so this pruning could be achieved by employing a closed list of visited states. However, in a temporal setting with disjunctive constraints it is not easy to check whether two partial plans with different sets of actions are equivalent. The pruning rule we presented manages at least to catch some of these cases.

Deleted Supports

A second pruning rule concerns pairs of actions being supported by the same proposition, whereas both actions

#	*Full*		*Older*		*-Pruning 1*		*-Pruning 1, Older*		*STP*	
	time	*makespan*	*time*	*makespan*	*time*	*makespan*	*time*	*makespan*	*time*	*makespan*
satellite1	1.1	177	1.7	269	81.83	1984	3.3	269	374.47	323
satellite2	10.4	453	8.8	182	-	-	13.88	182	-	-
satellite3	10.7	441	13.4	251	-	-	15.5	251	-	-
satellite4	43.1	501	-	-	-	-	-	-	-	-
satellite5	246.3	688	-	-	-	-	-	-	-	-

Figure 3: Preliminary results for various configurations of the planner (time in *secs*).

delete this proposition. Suppose there is a proposition P in the plan, being either an initial proposition or an add effect of an existing action. This specific proposition supports two actions A and B, for which $P \in Prec(A)$, $P \in Prec(B)$ *but also* $\neg P \in Eff(A)$ and $\neg P \in Eff(B)$ hold. It is obvious that this is an inconsistency and this partial plan must be discarded. However, the use of disjunctive constraints in our setting renders this inconsistency undetectable; it is detected only at the scheduling phase.

We implemented a mechanism in our system, which, each time a new causal link is created, checks which other existing actions are supported by the same proposition. For such pairs A and B of actions, being supported by the same proposition, the following constraints posted:

- If neither A nor B deletes P, no constraint is posted.
- If A deletes P but B preserves it, A is demoted after B.
- If B deletes P but A preserves it, B is demoted after A.
- If both A and B delete P, the plan is discarded.

Solution Generation and Presentation

Each time a partial plan is found, with *Agenda* being empty of open goals, a constraint solver is employed to search for feasible assignments of values to the constraint variables, i.e. the start times of the actions. Due to the extensive use of disjunctive constraints, the domains of the constraint variables are usually very broad before labeling begins. We use a branch and bound schema for solving the CSP problem, with the makespan of the plan being the minimized quantity. This ensures that we will get the shortest temporal plan, with respect to the specific partial plan at hand. Of course this is not the globally optimal plan, since alternative partial plans might result in even shorter makespans. If we want to find even better plans, we can continue the POCL planning procedure by setting the upper bound slightly shorter than the makespan of the found plan for the remaining partial plans.

Concerning the presentation of the solution plan, what we get from the CSP solver is an assignment of time points to the start times of the actions. If we want a more flexible presentation, we can present also the constraints between the actions. In case of disjunctive constraints, we can remove those disjuncts that are not satisfied by the current assignment. However, even with this removal, more than one disjuncts from each disjunction may be present, since, as we explicated earlier in this paper, the disjunctions do not correspond to mutually exclusive cases. Our current implementation keeps all these satisfiable disjuncts and presents them as part of the solution.

Preliminary Results

In this section we evaluate the various techniques we presented in the paper. We implemented a POCL temporal planner in Prolog and used ECLiPSe 5.8 as our environment. We ran our experiments in a Pentium 4, 3GHz, 1GB memory machine and we set a time limit of 5 mins. Our intention was not to create a high-performance planning system, so we did not give attention in implementation details but in declarativeness. As a result, our planner is not quite efficient to solve complex temporal problems.

We ran our system on the easiest of the temporal Satellite domain of the 4th International Planning Competition[1]. We compared the full featured system with alternative configurations, where one feature each time was deactivated. The results are shown in Figure 3. *Time* in all cases includes the creation of the temporal planning graph. We set a high enough upper bound for time, so all problems were solvable.

Columns *Full* present the performance of the full featured system. Columns *Older* present a configuration where ties are broken by preferring the older plans. We can see that in this case shorter plans are found, however more time is needed in general, making the larger problems unsolvable within the specified time limit.

Columns *STP* (for simple temporal problem) concern a configuration of the system, where threats were resolved by generating alternative children. This choice may increase dramatically the branching factor, so we applied it only to threats generated by emutex or permanent cmutex relations. In these cases, ties were broken by favoring promotion of new actions. As we can see, there was a significant increase in time needed to solve even the easiest problem.

Columns *–Pruning 1* concern the case where the repeated subgoal generation pruning rule was not used. We observe a significant negative impact in system's performance. In columns *–Prunning 1 older* ties are broken by preferring older plans. The results in this case are similar as in the *Older* case, although we receive slightly longer solution times.

We do not present results for the case where the second pruning rule was inactivated. In this case the planner failed

[1] The code and the problem files can be found at http://ai-server.uom.gr/pocl

to solve any problem, because the CSP subproblem generated by the planning procedure was unsolvable and detecting this unsolvability was very costly. Note that with the second pruning rule being active, no unsolvable CSP problem was generated in this domain.

We tried to run the system without considering threats generated by the temporary cmutex relations. In this case we did not notice any significant difference to the *Full* case. This was the case even when we tightened the upper bound. This is explained by the fact that large disjunctive constraints do not allow for propagation, especially in the presence of good heuristics. However, we think that the role of temporary cmutex relations needs further investigation, e.g. with a more powerful constraint propagation.

Finally we tried to run our system to problems from the Pipesworld and the Airport domain of the 4th Iternational Planning Competition. The results in this case were not consistent, being able to solve just a few instances occasionally. After careful investigation of this behavior we realized that it was due to early commitment in particular actions to support open goals, which was not easy to abandon at later stages. We believe that this problem could be alleviated with the use of lifted actions, which in turn raises new questions about the way lifted propositions could be evaluated by a heuristic function (e.g. taking the min or the average value from all their instantiations).

Conclusions and Future Work

In this paper we presented a temporal POCL planning system, which exploits a temporal planning graph in order to extract heuristic values and posts disjunctive constraints to resolve threats. We used a domain independent temporal heuristic that, among others, takes into account groups of permanently mutexed open goals and sums only the maximum of their heuristic values. Finally, we presented two completeness preserving pruning rules that are well suited in the framework of stratified (i.e. separated planning and scheduling) POCL temporal planning.

We believe that the use of disjunctive constraints is well suited in the framework of stratified heuristic POCL temporal planning, since it avoids the problem of overgenerating child plans with the same or similar heuristic values. On the other hand, disjunctive constraints demand for stronger propagation rules to exploit pruning challenges. In the future we aim to work at this direction.

References

Blum, A.L., and Furst,M.L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence*, 90 (1-2), 281-300.

Bonet, B., and Geffner, H. 2001. Planning as Heuristic Search. *Artificial Intelligence*, 129 (1-2), 5-33.

Bonet, B., Loerincs, G. & Geffner, H. (1997). A robust and fast action selection mechanism for planning. *In Proceedings of 14th National Conf. on Artificial Intelligence*, 714-719, Providence, RI. AAAI Press.

Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *Proc. of 2nd International Conf. on Artificial Intelligence Planning Systems*, 61-67, Chicago, IL: AAAI Press.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14, 253-302.

Joslin, D., and Pollack, M. 1994. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *Proc. of 12th National Conf. on AI*, 1004-1009, Seattle, WA. AAAI Press.

McAllester, D.A., and Rosenblitt, D. 1991. Systematic nonlinear planning. *Proceedings of Ninth National Conference on Artificial Intelligence*, 634-639. Anaheim, CA: AAAI Press.

Nguyen, X., and Kambhampati, S. 2000. Extracting effective and admissible heuristics from the planning graph. In *Proc. of 17th National Conf. on Artificial Intelligence*, 798-805, Austin, TX: AAAI Press.

Penberthy, J.S. and Weld, D.S. 1994. Temporal planning with continuous change. In Proc. of the 12th National Conference on Artificial Intelligence, 1010-1015, Seattle, WA: AAAI Press.

Penberthy, J.S., and Weld, D.S. 1992. UCPOP: A sound and complete, partial order planner for ADL. *In Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, 103-114. Cambridge, MA: Morgan Kaufmann Publishers.

Peot, M., and Smith, D. 1992. Conditional nonlinear planning. *In Proceedings of the first conf. on AI planning systems*, 189-197.

Peot, M., and Smith, D. 1993. Threat-removel strategies for partial order planning. *Proc. of the 11th National Conf. on AI*, 492-499.

Pryor, L., and Collins, G. 1996. Planning for contingencies: a decision-based approach. *Journal of Artificial Intelligence Research*, 4, 287-339.

Refanidis, I., and Vlahavas, I. 2001. GRT: Backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research*, 15 (2001), pp. 115-161.

Schubert, L., and Gerevini, A. 1995. Accelerating partial order planners by improving plan and goal choices. In *Proceedings of the 7th Intern. Conf. on Tools with Artificial Intelligence*, 442-450, Hernodn, VA. IEEE Computer Society Press.

Smith, D., and Weld., D. 1999. Temporal planning with mutual exclusion reasoning. *Proc. of the 16th Intern. Joint Conf. on Artificial Intelligence*, 326,333.

Smith, D.E., Frank, J., and Jonsson, A.K. 2000. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15(1), 47-83.

Vere, S.A. 1983. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(3), 246-267.

Vidal, V., and Geffner, H. 2004. Branching and Pruning: An Optimal Temporal POCL Planner based on Constraint Programming. In *Proc. 19th National Conference on Artificial Intelligence*, San Jose, CA: AAAI Press.

Younes, L.S.H., and Simmons, R.G. 2003. VHPOP: Versatile Heuristic Partial Order Planner. *Journal of Artificial Intelligence Research*, 20, 405-430.

Two Approaches to Semi-Dynamic Disjunctive Temporal Problems

Peter J. Schwartz and Martha E. Pollack

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI, USA
pschwart@eecs.umich.edu, pollackm@eecs.umich.edu

Abstract

This paper considers two approaches to improving efficiency and stability in semi-dynamic Disjunctive Temporal Problems (DTPs). A semi-dynamic DTP is a sequence of DTPs (Stergiou and Koubarakis 1998) in which each problem is a restriction of the problem before. We consider three basic types of DTP restrictions—tightening the bound of an existing constraint, adding a simple constraint, and adding a disjunctive constraint. The techniques we use for solving semi-dynamic DTPs come from the literature on dynamic Constraint Satisfaction Problems (CSPs) and include nogood recording (Schiex and Verfaillie 1993) and oracles (van Hentenryck and Provost 1991). Experimental results show that nogood recording improves efficiency but hurts stability, whereas oracles improve efficiency even more while also improving stability. The performance of nogood recording and oracles used in combination is not significantly different than the performance of oracles alone.

Introduction

The Disjunctive Temporal Problem (DTP) is a very expressive temporal constraint formalism (Stergiou and Koubarakis 1998). This expressivity comes at the cost of NP-complete complexity (Dechter, Meiri, and Pearl 1991), so a great deal of prior research has focused on heuristic approaches to solving DTPs (Stergiou and Koubarakis 2000, Tsamardinos and Pollack 2003, Armando et al. 2004). For many applications, however, it may be necessary to solve a *sequence* of DTPs in which each problem is very similar to the one before. Moreover, in at least some applications, it is much more common for one DTP in the sequence to be a restriction, rather than a relaxation, of the previous DTP in the sequence.

Execution-monitoring systems, such as Autominder (Pollack et al. 2003) and the Remote Agent (Muscettola et al. 1998), provide one example. Consider the Autominder system, a schedule maintenance and execution-monitoring system that issues reminders to keep a user on schedule. Autominder represents a user's schedule as a DTP, which is updated whenever time passes, an action is executed, or a new action and its constraints are added to the schedule. After each such change, Autominder must check the consistency of the new DTP. As long as the user neither retracts any constraints from the schedule nor executes an action that violates the schedule, each DTP in the sequence will be a restriction of the one before.

Plan generation systems that model complex temporal constraints, such as DT-POP (Schwartz and Pollack 2004) and TGP (Smith and Weld 1999), provide another example. DT-POP is a partial-order planner that can generate plans that include complex temporal relationships over multiple goals and actions, each of which may involve arbitrarily many time points. DT-POP represents all of these temporal relationships as DTPs, so each time a partial plan is modified to repair a flaw, DT-POP must solve a new DTP to determine whether the resulting plan is consistent. Whether adding an ordering constraint to repair a threat adding a causal link or new action to repair an open condition, the DTP is modified by adding or tightening constraints. As a result, the DTP of each partial plan in the search tree is a restriction of the DTP of its parent.

In this paper, we explore the question of solving *semi-dynamic DTPs*—sequences of DTPs in which each is a restriction of the previous one. We consider three types of basic DTP restrictions, which are jointly sufficient to represent any of the changes to a DTP described in the examples above. We employ two techniques that have been previously used for solving dynamic CSPs (sequences of finite-domain CSPs): nogood recording and oracles.

We are interested in how these techniques affect search efficiency as well as solution *stability*—i.e., the extent to which the solution to one DTP is similar to the solution to the previous DTP. As pointed out by Verfaillie and Schiex (1994, page 307), instability in a dynamic CSP "…may be unpleasant in the framework of an interactive design or a planning activity, if some work has been started on the basis of the previous solution." The same can be said of dynamic DTPs.

In the next section, we define DTPs and describe the standard approaches to solving them. We then introduce and formally define the semi-dynamic DTP. After that, we describe nogood recording and oracles, and we show how they can be applied to semi-dynamic DTPs. We then describe a set of experiments that compares these two techniques, both alone and together, and present results

showing their effect on efficiency and stability. We conclude with a discussion of the results and ideas for future work.

The Disjunctive Temporal Problem

The DTP is a generalization of the Simple Temporal Problem (STP). An STP (Dechter, Meiri, and Pearl 1991) is a pair $<V, C>$, where V is a set of temporal variables that represent time points and C is a set of simple temporal constraints over the time points of V. Each simple temporal constraint of C has the form $x - y \leq b$, where x and y are time points of V and b is a real number. Such a constraint is interpreted as "x follows y by no more than b units of time." An STP is often represented as a *d-graph*, in which each time point is a vertex and each constraint is a directed edge weighted by its bound. An STP is consistent iff its d-graph contains no negative cycles, so if $|V| = n$, then the STP can be solved in $O(n^3)$ time with a standard all-pairs shortest path algorithm such as Floyd-Warshall or Bellman-Ford (Cormen, Leiserson, and Rivest 1990).

A DTP is also a pair $<V, C>$, where V is a set of temporal variables that represent time points, but C is a set of *disjunctive* temporal constraints over the time points of V. Each disjunctive temporal constraint of C has the form $c_1 \vee c_2 \vee \ldots \vee c_m$, where each c_i is a simple temporal constraint. A disjunctive temporal constraint is satisfied if at least one of its component simple temporal constraints is satisfied. A DTP can express constraints that an STP cannot. For example, if a schedule contains two actions, A and B, that cannot overlap, a DTP can express this with the constraint "A ends before B begins, or B ends before A begins."

There are two predominant methods for solving a DTP. The first is to convert the DTP into a Satisfiability (SAT) problem. This method is used in TSAT++ (Armando et al. 2004), which is currently the fastest DTP solver. The second is to convert the DTP into a finite-domain CSP, as explained below. This method is used in Epilitis, which was the fastest when it was developed several years ago (Tsamardinos and Pollack 2003). Very little research has considered dynamic SAT problems (Hoos and O'Neill 2000), whereas much research has already been done on dynamic CSPs (van Hentenryck and Provost 1991, Schiex and Verfaillie 1993, Verfaillie and Schiex 1994). For this reason, the current work builds on the CSP method for solving DTPs.

To transform a DTP into a finite-domain CSP $<V, D, C>$ (called a *meta-CSP*), one has each variable of V represent a single disjunctive temporal constraint of the DTP, where the domain of each variable $v \in V$ is the set of component simple temporal constraints. The constraints C of the meta-CSP express the requirement that any combination of simple temporal constraints that is chosen as an assignment of the values as a solution of the meta-CSP must form a consistent STP.

For example, suppose we are given a DTP with time points $\{w, x, y, z\}$ and these constraints:

$c_1 = (z - y \leq 2)$
$c_2 = (w - y \leq 5) \vee (w - x \leq 7)$
$c_3 = (y - x \leq -3) \vee (y - w \leq -6)$

This DTP contains three disjunctive temporal constraints, so the corresponding meta-CSP will contain three variables. The domain of each variable in the meta-CSP is the set of component simple temporal constraints, so the domain of c_1 is $\{(z - y \leq 2)\}$, the domain of c_2 is $\{(w - y \leq 5), (w - x \leq 7)\}$, and the domain of c_3 is $\{(y - x \leq -3), (y - w \leq -6)\}$.

After this transformation, the DTP can be solved with any number of standard CSP techniques. Epilitis, the fastest DTP solver that uses this meta-CSP transformation, uses five different techniques to improve efficiency. Three of these techniques—forward checking, conflict-directed backjumping, and nogood recording—come from the finite-domain CSP literature. The other two techniques—removal of subsumed variables and semantic branching—are made possible by the fact that the temporal constraints are linear inequalities. See (Tsamardinos and Pollack 2003) for details on Epilitis.

The Semi-Dynamic DTP

As mentioned above, we define a semi-dynamic DTP to be a sequence of (static) DTPs, where each DTP in the sequence is a restriction of the previous one. The current work considers three types of restrictions in DTPs:

1. **tighten:** The bound b of a temporal constraint $x - y \leq b$ is reduced. When a bound is tightened, it is possible that some of the assignments in which it participates used to be valid but now create a negative cycle in the d-graph. The constraint that is tightened corresponds to a value of the meta-CSP, so tightening the bound of a constraint in the DTP has the effect of tightening a constraint in the meta-CSP.

2. **add STC:** A simple temporal constraint is added to the DTP. This corresponds to adding a variable to the meta-CSP with a domain size of one. The meta-CSP solver has no choice but to include the assignment of this value to this new variable in any potential solution that it considers.

3. **add DTC:** A disjunctive temporal constraint is added to the DTP. This corresponds to adding a variable to the meta-CSP with a domain size greater than one. The meta-CSP solver can choose any of the values in the domain of the new variable.

Some might argue for the inclusion of a fourth type of restriction—the removal of a disjunct from a disjunctive temporal constraint. In the meta-CSP, this would correspond to the removal of a value from the domain of a variable. It is possible to achieve the same effect by tightening the bound of the disjunct to negative infinity; even though the value is still present in the meta-CSP, it

could not participate in any solution and would be immediately pruned by forward checking. For this reason, we only consider the three types of restrictions listed above.

We can now formally define the semi-dynamic DTP:

Definition 1. A **semi-dynamic DTP** is a pair $<P_0, C>$, where P_0 is a static DTP, and C is a sequence of changes, each of which is one of the three types of restrictions listed above. If C contains n changes, then a semi-dynamic DTP solver must solve the sequence of DTPs $P_0, P_1, \ldots, P_n$, where each DTP P_i is created by restricting P_{i-1} according to change c_i, for $1 \leq i \leq n$.

The current work only considers semi-dynamic DTPs in which each change is a single restriction. In some systems, it might make more sense to perform several restrictions between subsequent DTPs. For example, if an event e is executed exactly t units of time after another event f, an execution-monitoring system can represent this by adding two simple temporal constraints to the schedule: $(e - f \leq t)$ and $(f - e \leq -t)$. Both of these constraints can be added before the new DTP is solved. Semi-dynamic DTPs in which each change is a set of restrictions remains an open avenue for future research.

We consider two techniques that have been shown to improve the performance of dynamic CSPs, adapting them to semi-dynamic DTPs. The next section describes nogood recording, and the section after that describes oracles.

Nogood Recording

Nogood recording (NGR) has been shown to be an effective technique for improving both static and dynamic CSPs (Schiex and Verfaillie 1993). Intuitively, a nogood is a partial assignment of variables that cannot be extended to a complete solution. Formally, a nogood of CSP $<V, D, C>$ (where V is the set of variables, D is the set of respective variable domains, and C is the set of constraints) is a pair $<A, J>$. A is an assignment to a subset of the variables of V, and J is a subset of V such that the constraints over J prevent A from participating in any solution (J is called the justification of the nogood).

As a CSP solver searches for a solution, it records a nogood each time it finds a dead end. As search continues, the CSP solver checks each partial assignment it considers against each nogood it has recorded. If it finds a nogood $<A, J>$ such that A is a subset of the partial assignment being considered, then that partial assignment can be pruned immediately. If the CSP solver is also using conflict-directed backjumping, then the justification J tells the solver how far it can safely backtrack.

In order to maximize the pruning power of each nogood, the assignment should be made minimal. This is accomplished by applying two basic properties of nogoods. First, if $<A, J>$ is a nogood, then $<A{\downarrow}J, J>$ is also a nogood, where $A{\downarrow}J$ is the projection of assignment A onto the variables of justification J. In other words, we can safely remove any variable in A that does not participate in the constraints over J. Second, if all possible extensions of an assignment A along a particular variable x have been recorded as nogoods $<A \cup \{x \leftarrow v_i\}, J_i>$ for all values v_i in the domain of x, then $<A, \cup_i J_i>$ is also a nogood. In other words, since any solution must assign a value to every variable, and since there is no valid extension of A that assigns a value to x, we know that A cannot be extended to a complete solution. During search, this second property makes it possible to record a nogood when backtracking over a non-leaf that is a generalization of all of its children combined.

Suppose we are solving a CSP with variables w, x, y, and z, all of which have the domain $\{1, 2, 3\}$. Say our current assignment is $\{w \leftarrow 1, x \leftarrow 2, y \leftarrow 3, z \leftarrow 1\}$, but this assignment is found to violate a constraint over w, x, and z. We could record the nogood $<\{w \leftarrow 1, x \leftarrow 2, y \leftarrow 3, z \leftarrow 1\}, \{w, x, z\}>$, but the first property of nogoods tells us that we can instead record the smaller nogood $<\{w \leftarrow 1, x \leftarrow 2, z \leftarrow 1\}, \{w, x, z\}>$. This nogood has greater pruning power because it applies no matter what value is assigned to y. Now say that, after further search, we discover that $<\{w \leftarrow 1, x \leftarrow 2, z \leftarrow 2\}, \{w, x, z\}>$ and $<\{w \leftarrow 1, x \leftarrow 2, z \leftarrow 3\}, \{w, x, z\}>$ are also nogoods. At this point, we have seen that the assignment $\{w \leftarrow 1, x \leftarrow 2\}$ extended with any assignment to z leads to a dead end, so the second property of nogoods tells us that we can record the nogood $<\{w \leftarrow 1, x \leftarrow 2\}, \{w, x, z\}>$. This nogood has more pruning power because it applies no matter what value is assigned to z.

Epilitis was the first system to apply NGR to static DTPs. The only difference between applying NGR to finite-domain CSPs and DTPs comes when finding the justification of a nogood at a leaf dead end. In a CSP, the constraints are given in some explicit description, but in the meta-CSP of a DTP, the constraints are given implicitly that any assignment must form a consistent STP. An STP is inconsistent iff its d-graph contains a negative cycle, so the justification of a nogood in a DTP is found by simply identifying the time points involved in the negative cycle, and then identifying the meta-CSP variables whose assignments created that negative cycle.

In practice, the number of nogoods that are recorded can grow to an unmanageable size, so a few basic techniques can be used to prevent this from happening. First, as the reader may have noticed while looking at the previous example, when the second property of nogoods is applied, the new nogood that is generated is a generalization of the nogoods that were combined to create it. Whenever the second property is applied, all of the nogoods that were combined to create the new one can be removed from the set of nogoods without losing any pruning power. Second, the more variables in a nogood's assignment, the fewer assignments they apply to, so nogoods with large assignments have little pruning power. It is therefore common practice to select a size limit and only record nogoods whose assignment is within this limit. Tsamardinos and Pollack (2003) suggest a nogood size

limit of 10 for Epilitis, and we follow their suggestion in our implementation.

The application of NGR to dynamic problems is straightforward. Nogoods are recorded while searching for a solution to one problem instance and then applied while searching for a solution to the next problem instance. If the problem is relaxed, some of the nogoods may no longer be valid and would have to be removed from the set. The current work, however, only considers semi-dynamic DTPs in which no relaxations are permitted, so we do not need to deal with this case. A restriction only removes solutions of a problem instance, so any partial assignment that could be safely pruned by a nogood before the restriction can be safely pruned after.

Oracles

Oracles (van Hentenryck and Provost 1991) have also been applied to dynamic finite-domain CSPs. During CSP backtracking search, all partial assignments that were tested before the first solution was found must have been pruned. After a restriction to the CSP, those same partial assignments can be immediately pruned from the new search space. An oracle records the path to the first solution so the next search can try to repeat it, bypassing the pruned portion of the search space.

Formally, we define an oracle for a problem as a triple <*A*, *O*, *P*>. *A* is the solution assignment of the *previous* problem in the sequence, *O* is the order in which variables were chosen to find this solution, and *P* is the set of values that were tested and pruned from the domain of each variable along the way. If the current problem resulted from a restriction, then all of the values in *P* can be safely pruned again without being tested. If the solver is also using conflict-directed backjumping or NGR, then a justification must be recorded along with each pruned value of *P*.

Given a CSP <*V*, *D*, *C*> and an oracle <*A*, *O*, *P*>, backtracking search proceeds as follows. First, a variable *var* is selected according to *O*. Each value in the domain of *var* that appears in *P* can be pruned immediately without search. Then the value assigned to *var* according to *A* is chosen. This process continues until either a new solution is found or the value that should be chosen according to *A* causes an inconsistency. In the latter case, the oracle is abandoned from that point in the search. From then on, variables and values are selected according to the solver's heuristics, and no more values are pruned without first being searched. If a solution is found, a new oracle can be generated from the search path to guide the next search.

An oracle can be generated directly from a search tree once the first solution is found. Consider the search tree shown in Figure 1. The nodes that are crossed out represent partial assignments that were tested and pruned, the black nodes represent partial assignments along the search path to the solution, and the white nodes represent partial assignments that have not been tested yet. The oracle is enclosed in a dotted line.

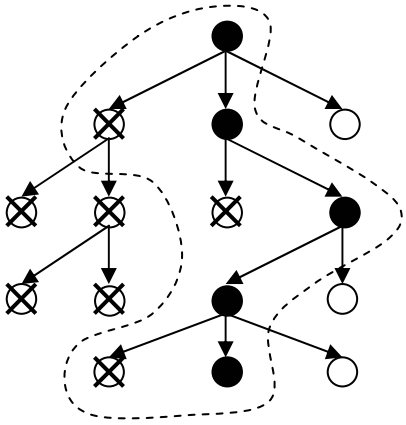


Figure 1. An oracle from a CSP search tree.

The pseudo-code for solving a CSP with an oracle appears in Figure 2. The **Oracle-Search** procedure takes three arguments: the CSP <*V*, *D*, *C*> that is currently being solved, the oracle <*A*, *O*, *P*> from the previous problem that is guiding the current search, and the oracle <*A'*, *O'*, *P'*> that is being created to guide the next search. If the given CSP is consistent, **Oracle-Search** returns the new oracle (which contains the solution assignment); otherwise, **Oracle-Search** returns FAIL.

Oracle-Search is a recursive procedure, so it begins by checking if all of the variables have been assigned, and, if so, it returns the new oracle (line 1). Otherwise, it selects the next variable to be assigned. If the old oracle is still being followed, **Oracle-Search** follows the variable ordering *O* from the previous search, and all of the values that were tested and pruned during the previous search can be pruned immediately without being tested in the current search (lines 3-6). If the old oracle has been abandoned, **Oracle-Search** falls back on a heuristic variable selection method (lines 7-8). Either way, the new variable choice is appended to the variable ordering of the new oracle (line 9).

Now **Oracle-Search** tests the values in the working domain of the chosen variable until one is found that leads to a solution or all of them have failed. The first step is to choose a value to test. If the old oracle is still being followed, **Oracle-Search** starts by testing the value that was assigned to the chosen variable in the previous solution assignment (lines 11-12). If the old oracle has been abandoned, **Oracle-Search** falls back on a heuristic value selection method (lines 13-14). The chosen value is removed from the working domain of the chosen variable (line 15).

Once the variable and value have been chosen, the new binding is added to the current assignment and tested against the constraints (lines 16-17). If none of the constraints are violated, **Oracle-Search** calls itself recursively (line 18). In this recursive call, the new oracle has been modified to reflect the new variable and value choices, along with any other values that have been pruned from the chosen variable's working domain. If the chosen

value leads to a solution, then that solution (which is the new oracle) is returned immediately (line 19).

If the chosen value does not lead to a solution, or if the new assignment failed the constraint check, then the value is pruned from the working domain of the chosen variable (line 20). The fact that this value has been pruned is recorded in the new oracle (line 21). Since the value from the previous solution assignment is always tested first, that value must have failed by this point. This means that the oracle cannot lead the solver directly to a solution, so the oracle is abandoned (line 22). From this point on, **Oracle-Search** will behave exactly like a standard CSP backtracking search algorithm, except that it will continue to build the new oracle to guide the next search. If all of the values in the working domain of the chosen variable have failed to lead to a solution, then **Oracle-Search** returns FAIL (line 23).

```
Oracle-Search(<V, D, C>, <A, O, P>, <A', O', P'>)
// <V, D, C> is the CSP being solved
// <A, O, P> is the oracle from the previous problem
// <A', O', P'> is the new oracle being created
1  if all variables assigned, then return <A', O', P'>
2  P" ← P'
3  if still following old oracle <A, O, P>, then
4     choose next variable var according to O
5     prune all values from D[var] that appear in P[var]
6     P"[var] ← P[var]
7  else (old oracle has been abandoned)
8     choose next variable var according to heuristic
9  let new ordering O" be O' appended with var
10 while working domain D[var] is not empty
11    if still following old oracle <A, O, P>, then
12       choose value val according to A[var]
13    else (old oracle has been abandoned)
14       choose value val according to heuristic
15    remove val from D[var]
16    let new assignment A" be A' ∪ {var←val}
17    if A" does not violate constraints C then
18       sol ← Oracle-Search(<V, D, C>, <A, O, P>,
             <A", O", P">)
19       if sol ≠ FAIL then return sol
20    prune val from D[var]
21    P"[var] ← P"[var] ∪ {val}
22    abandon the old oracle
23 return FAIL
```

Figure 2. A CSP solver that uses an oracle to guide search.

We can make several observations about the behavior of this algorithm. If the solution of the previous problem is still a valid solution after the restriction, then the oracle will lead the solver directly to it. If a new variable has been added to the problem, the oracle will cause the solver to try to extend the previous solution by assigning the new variable each of the values in its domain before it backtracks to test any other assignments. If the solution of the previous problem is no longer a valid assignment after the restriction, then the oracle will still force the solver to use as much of the previous solution as possible.

The use of oracles can be applied directly to the meta-CSPs of a semi-dynamic DTP. As with nogoods, a relaxation could invalidate some of the justifications that are recorded with the pruned values of the oracle, so these values would have to be removed from the pruned set before the next search was to begin. The current work, however, only considers restrictions, so we do not need to deal with this case.

Experimental Comparison

We compare the efficiency and stability of four algorithms for solving semi-dynamic DTPs. Since nogood recording and oracles operate on CSPs and Epilitis is the fastest algorithm to solve DTPs using the meta-CSP transformation, we use Epilitis as the underlying static DTP solver of all of our algorithms.

As stated earlier, TSAT++ is currently the fastest static DTP solver, but the publicly available implementation of TSAT++ only outputs four pieces of information after solving a DTP: the consistency of the given DTP, the total time TSAT++ used to determine consistency, the search time, and the number of constraint checks performed. It does not output a solution of the DTP when it is found to be consistent. This makes it impossible to reuse solutions or to measure the stability of solutions of subsequent DTPs in the sequence, so a fair comparison to TSAT++ is not possible at this time. We are presently working with the creators of TSAT++ to get an implementation that does output solutions.

The four algorithms we tested are as follows:

1. **Naïve:** The first algorithm is a naïve algorithm that repeatedly applies Epilitis to solve each DTP in the sequence from scratch. Even though this algorithm (as well as all of the others) uses nogood recording as a method to enhance the efficiency of Epilitis, the nogoods learned during one search are forgotten when the DTP changes.

2. **NGR:** The second algorithm is exactly the same as the first algorithm, except that all of the recorded nogoods are applied to all subsequent DTPs in the sequence.

3. **OR:** The third algorithm forgets all of its recorded nogoods when the DTP changes just like the naïve algorithm does, but this algorithm generates an oracle after each DTP is solved and applies it when solving the next DTP in the sequence.

4. **Both:** The fourth algorithm combines both techniques. It applies recorded nogoods to all subsequent DTPs, and it generates an oracle after solving one DTP and applies it to the next.

As pointed out in the section on oracles, if the previous solution is still a valid assignment after the restriction, an oracle will guide the search to test the previous solution

first and then extend it if a new meta-CSP variable has been added to the problem. To facilitate better comparison, we apply this same technique to both of the algorithms that do not use oracles. Before starting a new search, all of the algorithms check whether the previous solution is still valid, or if it can be extended when a new variable has been added to the meta-CSP. A new search is performed only if the previous solution is not (and cannot be extended to) a solution of the current problem.

To compare the algorithms, we generated four sets of 50 semi-dynamic DTPs. Each semi-dynamic DTP consists of a consistent initial DTP (P_0) and a sequence of restrictions (C). The first set contains only tighten changes, the second set contains only add STC changes, the third set contains only add DTC changes, and the fourth set contains all three types of changes chosen from a uniform distribution. The sequence of changes terminates when either the DTP becomes inconsistent (because more restrictions will always produce inconsistent DTPs) or the DTP has been changed 50 times.

The initial DTP of each dynamic problem is generated randomly based on the parameters of the most difficult problems considered in (Tsamardinos and Pollack 2003). Each of the initial DTPs in our experiments contains 30 time points and 180 disjunctive temporal constraints. This is a ratio of 6 disjunctive constraints to each time point, which Tsamardinos and Pollack found to be a phase transition point where DTPs are the most difficult to solve. Each disjunctive temporal constraint is the disjunction of 2 simple temporal constraints, and the bound of each simple constraint is chosen randomly with uniform probability from the integers of [-100, 100].

A tighten change is generated by randomly selecting a simple constraint of one of the disjunctions and reducing its bound by a random integer amount in [0, 100]. An add STC change is generated in the same manner as the disjunctive constraints of the initial DTP, except that the new constraint contains only 1 disjunct instead of 2. An add DTC change is generated in exactly the same manner as the disjunctive constraints of the initial DTP. For all types of restrictions, the set of time points in the DTP does not change.

Each type of change affects the DTP to a different degree. A tighten change only has a probability of 0.5 of tightening a disjunct that is part of the solution from the previous problem, and even if it does, it is still possible that the bound is not reduced enough to invalidate the previous solution. An add STC change forces the solution to include one particular simple constraint, so it is more likely to invalidate the previous solution. An add DTC change forces the solution to include a new simple constraint, but the particular simple constraint can be chosen from the set of (two) disjuncts in the new disjunctive constraint. Because of the inherent differences among the types of changes, the semi-dynamic DTPs generated by applying them differ in both the average number of changes that are made before the problem becomes inconsistent and the percentage of solutions that can be reused for each algorithm. Table 1 shows the differences among the four problem sets. The values in Table 1 demonstrate that the average number of changes and percentage of reused solutions can both vary greatly with the change type, but vary little among the different algorithms.

change type	**avg number of changes before inconsistent**	**% solutions reused**			
		Naïve	**NGR**	**OR**	**Both**
tighten	39.6	91.91	92.32	92.27	92.22
add STC	5.1	62.75	61.57	61.66	61.57
add DTC	16.0	82.75	82.38	82.20	81.75
all	13.4	78.81	78.81	80.06	79.25

Table 1. Average number of changes per problem and percentage of solutions reused for semi-dynamic DTPs with different types of restrictions.

We ran each of the four algorithms on all of the problems and measured the efficiency and stability of each algorithm. Efficiency was measured in two ways: the CPU time and the number of nodes (i.e., partial assignments) tested while solving each semi-dynamic DTP. Stability is measured as the percent of variable bindings from the previous solution that match the new solution. Recall that a new search is only necessary when the previous solution cannot be reused or extended. If all of the same variable bindings from the previous solution match the new solution, then the stability is 100%; however, in the case of 100% stability, a new search is not necessary, so these cases are not included in the averages that are reported. Thus, the stability values reported are lower than they would be had the solution-reuse cases been incorporated into the averages. Tables 2, 3, 4, and 5 show the results of tighten, add STC, add DTC, and all changes, respectively. The algorithms were all implemented in Java and the experiments were run on a 3.0 GHz Pentium 4 machine with 1 GB of memory.

	Naïve	**NGR**	**OR**	**Both**
time (sec)	37.60	19.38	11.91	12.86
nodes	27,777	14,041	9,808	10,116
stability	75.50	73.60	79.20	79.40

Table 2. Efficiency and stability on tighten restrictions.

	Naïve	**NGR**	**OR**	**Both**
time (sec)	13.44	11.99	9.79	10.39
nodes	9,778	8,863	7,808	8,230
stability	73.46	67.15	75.10	75.80

Table 3. Efficiency and stability on add STC restrictions.

	Naïve	NGR	OR	Both
time (sec)	39.14	28,69	21.86	22.09
nodes	27,882	20,596	16,452	16,309
stability	76.80	74.18	80.40	80.17

Table 4. Efficiency and stability on add DTC restrictions.

	Naïve	NGR	OR	Both
time (sec)	16.48	12.39	9.45	10.25
nodes	12,037	8,953	7,770	8,285
stability	73.30	71.67	76.10	76.32

Table 5. Efficiency and stability on all restrictions.

Discussion

Tables 2-5 show that for every type of restriction, nogood recording helps efficiency but hurts stability relative to the naïve approach. NGR helps efficiency because the nogoods allow earlier pruning in DTPs that appear later in the sequence. NGR can hurt stability because Epilitis counts the number of nogoods in which each value participates as part of its variable and value selection heuristics, so the additional nogoods can steer the DTP solver to a very different part of the search space. Even if the number of nogoods for each value was not incorporated directly into the variable and value selection heuristics, the additional nogoods from the previous search would still allow forward checking to prune some values early. Since most heuristics count the number of values in the working domain of each variable, and since the additional nogoods would remove some of these values, we would still expect the heuristics to guide the search to a different part of the search space, leading to a reduction in stability.

Tables 2-5 also show that for every type of restriction, the use of oracles resulted in better efficiency and stability than either the naïve approach or nogood recording. On one hand, an oracle can improve efficiency by pruning part of the search space and reusing part of the previous solution. On the other hand, an oracle might have hurt performance by leading the solver to part of the search space where it will not find a solution. Apparently, oracles help performance much more than they hurt it: they improve performance in all cases, most dramatically reducing CPU time to less than one third of the naïve approach when tightening bounds (see Table 2). Oracles improve stability by starting the search from the previous solution, generally increasing stability by 2 to 3 percent over the naïve approach. This is in contrast to nogood recording, which generally decreased stability by roughly the same amount.

It is interesting to compare the performance of the combination of oracles with nogood recording against the performance of oracles alone. We had expected that the combination of techniques, when compared to oracles alone, would result in equivalent stability (because the oracles would still guide the beginning of the search) but improved efficiency (because the nogoods would allow earlier pruning and provide more informed heuristics when the oracle is abandoned). Tables 2-5 show us that, overall, the performance of the combination of techniques was nearly equivalent to the performance of oracles alone in both stability (as we had expected) and efficiency (as we had *not* expected). We conjecture that the efficiency of oracles alone versus oracles with nogood recording is so similar because the stability resulting from oracles is so high (generally in the range of 75 to 80 percent); this means that only a relatively small number of variables in the meta-CSP need to be reassigned, so the additional pruning and heuristic power gained through the use of nogoods has little opportunity to influence efficiency. Additional experimentation is necessary to determine whether or not this is true.

Conclusions

This paper is an initial attempt to improve efficiency and stability in semi-dynamic DTPs. We have presented two approaches and examined their effectiveness. Our initial experimental results show that oracles are more effective than nogood recording at improving both efficiency and stability. We hope to continue our research by running more experiments that vary the number of time points and the ratio of time points to disjunctive temporal constraints, and by comparing the techniques on real data.

There are many possible ways to extend this work. This work only investigates restrictions on DTPs, so one obvious extension is to consider the problem of fully dynamic DTPs, which include both restrictions and relaxations. We already touched briefly on how relaxations can complicate both nogood recording and oracles. Another extension would be to look at other dynamic CSP algorithms and apply them to dynamic DTPs. For example, the local changes algorithm of (Verfaillie and Schiex 1994) strives to reuse as much of the previous solution as possible, so it might be able to improve stability, but possibly at the cost of efficiency.

Acknowledgements

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA), through the Dept. of the Interior, NBC, Acquisition Services Division, under Contract No. NBCH-D-03-0010, and on work supported by the Air Force Office of Scientific Research, under Contract No. FA9550-04-1-0043.

References

Armando, A., Castellini, C., Giunchiglia, E., and Maratea, M. 2004. A SAT-Based Decision Procedure for the Boolean Combination of Difference Constraints. *Proceedings of SAT04*.

Cormen, T., Leiserson, C., and Rivest, R. 1990. *Introduction to Algorithms*. Cambridge, Mass.: MIT Press.

Dechter, R., Meiri, I., and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61-95.

Hoos, H. and O'Neill, K. 2000. Stochastic Local Search Methods for Dynamic SAT—an Initial Investigation, Technical Report TR-00-01, Computer Science Department, University of British Columbia.

Muscettola, N., Nayak, P., Pell, B., and Williams, B. 1998. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence* 103(1-2):5-48.

Pollack, M.E., Brown, L., Colbry, D., McCarthy, C.E., Orosz, C., Peintner, B., Ramakrishnan, S., and Tsamardinos, I. 2003. Autominder: An Intelligent Cognitive Orthotic System for People with Memory Impairment. *Robotics and Autonomous Systems* 44:273-282.

Schiex, T., and Verfaillie, G. 1993. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools* 3(2):187-207.

Schwartz, P., and Pollack, M.E. 2004. Planning with Disjunctive Temporal Constraints. *ICAPS-04 Workshop on Integrating Planning into Scheduling*.

Smith, D., and Weld, D. 1999. Temporal Planning with Mutual Exclusion Reasoning. *Proceedings of the 16th International Joint Conference on Artificial Intelligence*.

Stergiou, K, and Koubarakis, M. 1998. Backtracking Algorithms for Disjunctions of Temporal Constraints. *15th National Conference of Artificial Intelligence (AAAI-98)*.

Stergiou, K, and Koubarakis, M. 2000. Backtracking Algorithms for Disjunctions of Temporal Constraints. *Artificial Intelligence* 120:81-117.

Tsamardinos, I., and Pollack, M. 2003. Efficient Solution Techniques for Disjunctive Temporal Reasoning Problems. *Artificial Intelligence* 151(1-2):43-90.

van Hentenryck, P., and Provost, T.L. 1991. Incremental Search in Constraint Logic Programming. *New Generation Computing* 9:257-275.

Verfaillie, G., and Schiex, T. 1994. Solution Reuse in Dynamic Constraint Satisfaction Problems. *Proceedings of AAAI-94*.

Exploiting the Structure of Hierarchical Plans in Temporal Constraint Propagation

Neil Yorke-Smith and **Mabry Tyson**
Artificial Intelligence Center, SRI International, Menlo Park, CA 94025, USA
{nysmith,tyson}@ai.sri.com

Abstract

Quantitative temporal constraints are an essential requirement for many planning domains. The HTN planning paradigm has proven to be better suited than other approaches to many applications. To date, however, integrating temporal reasoning with HTN planning has been little explored. This paper describes a means to exploit the structure of a HTN plan in performing temporal propagation on an associated Simple Temporal Network. By exploiting the natural restriction on permitted temporal constraints, the time complexity of propagation can be sharply reduced, while completeness of the inference is maintained. Empirical results indicate an order of magnitude improvement on real-world plans.

Introduction

Quantitative temporal constraints are an essential requirement for many real-life planning domains (Smith, Frank, & Jónsson 2000). The Hierarchical Task Network (HTN) planning paradigm has proven to be better suited than other approaches to many applications (Myers *et al.* 2002). To date, however, integrating temporal reasoning within the HTN planning process has been explored in only a few systems.

This paper describes a means to exploit the structure of a HTN plan in performing temporal propagation on an associated Simple Temporal Network (STN). We introduce an algorithm called *sibling-restricted propagation* that exploits the restricted structure of STNs that arise from an HTN plan. The idea behind the algorithm is to transverse a tree of sub-STNs that correspond to the expansions in the HTN task hierarchy. The HTN structure limits the sub-STNs to have constraints only between parent and child nodes and between sibling nodes. Because the STNs thus considered are small, compared to the *global* STN corresponding to the whole plan, the overall amount of work to perform propagation is sharply reduced. Empirical results demonstrate an order of magnitude improvement on real-world plans.

Many metric temporal planners adopt an STN (or its generalisation to a Disjunctive Temporal Network) to describe the temporal relations underlying the plan. HTN planners in this category include O-Plan (Tate, Drabble, & Kirby 1994), SIPE-2 (Wilkins 1999), HSTS/RA/Europa (Jonsson *et al.* 2000), PASSAT (Myers *et al.* 2002), and SHOP2 (Nau *et al.* 2003). Similar representations are used by other HTN systems, such as IxTeT (Laborie & Ghallab 1995). Our work is distinguished by explicit use of the HTN plan structure to propagate on the underlying STN. Of the systems that do not adopt an STN, temporal HTN planning has been addressed both from a classical HTN planning perspective, such as Yaman & Nau (2002), and from other perspectives, such as logic programming (Son, Baral, & Tuan 2004).

Combining planning and scheduling has been approached from both sides of the gap (Smith, Frank, & Jónsson 2000). Specific algorithms have been developed for temporal propagation (e.g. Tsamardinos, Muscettola, & Morris (1998)) and resource propagation (e.g. Laborie (2003)) within a planning context. Again, while numerous systems feature methods to efficiently propagate temporal information and use it in the planning or scheduling process, we are not aware of any published results on specific algorithms to exploit HTN structure in STN propagation.

Whether in the HTN paradigm or not, it is possible to encode the whole temporal planning problem as a Constraint Satisfaction Problem — as done by El-Kholy & Richards (1996), Do & Kambhampati (2001), Mali (2002), and Frank & Jónsson (2004), among others. For us, the benefit of closely integrating casual task inference and STN-based task scheduling in the overall reasoning process, while maintaining distinction between them, comes in exploiting the specialised nature of the two aspects.

The next section presents necessary background on HTN planning, Simple Temporal Networks, and STN propagation algorithms. The following sections introduce the sibling-restricted propagation algorithm, present an initial characterisation of its theoretical properties, and evaluate its implementation in the PASSAT plan authoring system.

Background

Hierarchical Task Network planning (Erol, Hendler, & Nau 1994) generalises traditional operator-based planning through the addition of methods. Methods encode rich networks of tasks that can be performed to achieve an objective. Tasks within a method are temporally partially ordered, and may have associated preconditions and effects in addition to those of the method as a whole. With HTN methods, planning can assume a hierarchical flow, with high-level tasks

being decomposed progressively into collections of lower-level tasks through the application of matching methods with satisfied preconditions. Most large-scale, realistic planning applications to date have employed the HTN paradigm (Smith, Frank, & Jónsson 2000).

Simple Temporal Networks For modelling and solving the temporal aspects of planning and scheduling problems, quantitative temporal constraint networks in the form of the Simple Temporal Problem (STN) (Dechter, Meiri, & Pearl 1991) are widely adopted. Schwalb & Vila (1998) survey the wider work on temporal constraint satisfaction, including qualitative and hybrid formulations.

An STN is a restriction of the *Temporal Constraint Problem* to have a single interval per constraint. Variables X_k denote time-points and constraints represent binary quantitative temporal relations between them. A distinguished time-point, denoted *TR*, marks the start of time. Unary domain constraints are modelled as binary relations to *TR*; thus all constraints have the form: $l_{ij} \leq X_j - X_i \leq u_{ij}$, where l_{ij} and u_{ij} are the lower and upper bounds respectively on the temporal distance between time-points X_i and X_j, i.e. $X_j - X_i \in [l_{ij}, u_{ij}]$.

Consistency of an STN can be determined by enforcing path consistency on the distance graph arising from the constraints (Dechter, Meiri, & Pearl 1991). Moreover, an STN, together with the *minimal network* of time-point domains, can be specified by a complete directed graph, its *d-graph*, where edge $i \rightarrow j$ is labelled by the shortest path length d_{ij} between X_i and X_j in the distance graph. Any All-Pairs Shortest Path algorithm (e.g. Floyd-Warshall) may be used to compute the d-graph given the distance graph, and the d-graph may be represented as a sparse or dense *distance matrix*. We denote its computation by PC.

Like many other planners, PASSAT employs an STN to represent the temporal aspects of plans, using an approach called constraint-based interval planning (Frank & Jónsson 2004). The temporal extent of each task is modelled by a time-point each for its start and end. At regular occasions in the planning process, checking consistency of the temporal constraints and propagation of temporal information is required. This is achieved by invoking PC on the plan's STN.

Propagation The basic method for PC is to use an All-Pairs Shortest Path algorithm on the distance matrix A; the STN is consistent iff no diagonal element is negative ($a_{ii} < 0$ for some i corresponds to a cycle in the d-graph) (Dechter, Meiri, & Pearl 1991). In the terminology of Bessière (1996), this method is PC-1. Let the d-graph have V vertices and E edges. The complexity of PC-1 is $\Theta(V^3)$ (Floyd-Warshall) for a dense representation of the graph and $\Theta(V^2 \log V + VE)$ (Johnson's algorithm) for a sparse representation. Note that a HTN with n tasks has $2n + 1$ vertices in the d-graph of its STN: two time-points for each task, plus one for *TR*.

For incremental recomputation when a constraint is added or removed, a Single-Source Shortest Path algorithm can be used (Cesta & Oddi 1996). The complexity is $\Theta(VE)$ (Bellman-Ford) for a dense graph and $\Theta(E + V \log V)$ (Dijkstra) for a sparse graph.

Dechter (2003) and Bessière (1996) present other path consistency algorithms that can be specialised for the STN and used to implement PC computation. Of note is PC-2, which avoids redundant computation by use of an auxiliary data structure. For an STN with n time-points, if PC-1 is $\Theta(n^3)$ time and $\Theta(1)$ space, PC-2 is $\Theta(n^3)$ time but $\Theta(n^3)$ space, but exhibits better performance in practice provided the space requirements do not dominate (Bessière 1996). Dechter (2003) also presents an algorithm DPC that determines consistency but does not obtain the minimal network; separately, Cesta & Oddi (1996) present an incremental algorithm with the same function.

These algorithms are largely subsumed by △STP (Xu & Choueiry 2003). This algorithm, which does find the minimal network, outperforms PC-1, and is comparable to (dense graphs) or outperforms DPC (sparse graphs). The algorithm proposed in this paper invokes an STN solver repeatedly on different STNs; △STP or any of the other methods described for PC may be employed.

Sibling-Restricted Propagation

The idea behind sibling-restricted propagation is to exploit the HTN structure, under a mild restriction on permitted temporal constraints. Simple temporal constraints are permitted only between parent tasks and their children, and between sibling tasks. For example, suppose task **A** has been decomposed into tasks **B** and **E**. Temporal constraints are permitted between the start and end time-points of **A**, **B** and **E**. They are permitted between **B** and its children, but not between **A** or **E** and the children of **B**. Temporal constraints are also prohibited between **B** and some other task **X**.

This assumption on what STN constraints may exist between plan elements is inherent to HTN models. In particular, there is no way in standard HTN representations to specify temporal constraints between tasks in different task networks. Thus sibling-restricted propagation imposes no additional limitations on the expressiveness of HTNs.

The STN that arises from an HTN with the sibling constraint restriction has marked structure properties. The STN can be decomposed into a tree of smaller STNs; the shape of this tree mirrors the shape of the hierarchical structure in the plan. By traversing this tree, invoking PC at each 'node' STN, we can propagate temporal information on the plan elements. The restriction on constraints guarantees we can propagate on this tree and lose no information compared to propagating with the whole global STN: it means that the algorithm SR-PC presented below is sound and complete.

Expanding a task τ into its children imposes some implied HTN constraints. Namely, each child τ_i cannot start before or finish after its parent; in terms of Allen's interval algebra (Allen 1983), τ_i `during` τ. STN constraints can represent all of Allen's relations. They can also represent the partial ordering of children in a task network, which we denote $\tau_i \leq \tau_j$ (of course, children need not be ordered). What cannot be expressed are disjunctive constraints, such as task non-overlap, i.e. "τ_i occurs before or after τ_j".

Algorithm Description

To explain the SR-PC algorithm we need some details on the distance matrix representation A of a set of tasks. The

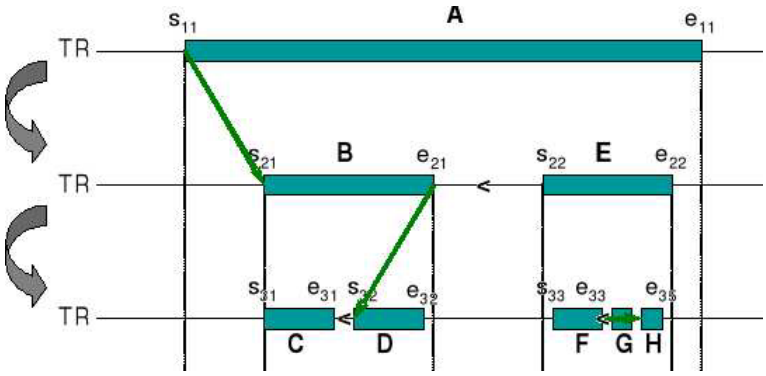


Figure 1: Example HTN plan with two levels of expansion

domain of a time-point — its current known earliest possible start and latest possible finish — is given by its current temporal distance from *TR*. In the minimal network form of the STN, the domain of every time-point is the broadest possible, given the constraints, such that every value in the domain participates in at least one feasible solution to the STN (Dechter, Meiri, & Pearl 1991). Without loss of generality, we order our distance matrices with $TR = X_0$ as the first time-point, i.e. the first row and column. Then the domain of a time-point X_i is $[-a_{i0}, a_{0i}]$. The initial domain of X_i is given by any constraints between it and *TR*; if there are none, its default initial domain is $(-\infty, \infty)$.

Secondly, the duration of a task τ is given the bounds on the distance between its start and end time-points (let them be X_i and X_j), i.e. the interval given by the minimum and maximum possible temporal distances between them.[1] If there is an explicit constraint between X_i and X_j, we call the temporal distance it describes the *local domain* of τ. For example, the constraint $10 \leq X_j - X_i \leq 20$, implies that τ's local domain is $[10, 20]$. The local domain is a bound on τ's duration, possibly not tight if the STN is not minimal. Again without loss of generality, we order our distance matrices to pair the start and end time-points of each task, so that the k^{th} task is modelled by time-points X_{2k-1} and X_{2k}. Then the local domain of task τ_k is $[-a_{2k,2k-1}, a_{2k-1,2k}]$.

Bounds on the duration of τ can be computed in a second way: from the domains of X_i and X_j, provided it is possible to relate these time-points to *TR*, i.e. their domains are more informative than $(-\infty, \infty)$; we call this the *global domain* of τ. If the plan has no temporal constraints that relate τ to *TR*, and the user has not specified when τ starts or ends (in absolute time or relative to *TR*), then the global domain on τ will be computed as $(-\infty, \infty)$. However, if X_i and X_j are related to *TR*, then after PC is complete the duration of τ computed from their domains will coincide with the local domain of τ. In general, the duration of τ is contained in the intersection of the two sets of bounds, local and global.

Pseudocode for SR-PC is shown in Algorithm 1. Given a task in a HTN plan, which we call the *root* task τ for the invocation, the algorithm updates the durations of the task and all its descendents in the plan, by recursively following the HTN expansions. Note the root task need not be the top-level objective of the plan, i.e. the root of the whole HTN.

Algorithm 1 Sibling-Restricted Propagation

```
1:  SR-PC (TR, root task τ)
2:  if τ is not a leaf node then
3:      Create distance matrix A for τ
4:      Perform PC on A, and update domains
5:      L ← children of τ                          {list of pending child nodes}
6:      repeat
7:          for each child c ∈ L do
8:              SR-PC (TR, c)                      {recurse}
9:              Update local domain of c in A
10:         if any change to A occurred then
11:             Perform (incremental) PC on A and update domains
12:         L ← ∅
13:         for each child c of τ do
14:             if c's global or local domain changed by line 11 then
15:                 Add c to L                     {must reconsider c}
16:     until L = ∅
17: return
```

Provided τ is not a leaf in the plan hierarchy, i.e. is not a primitive action or an unexpanded task, we create a distance matrix A (line 3). The time-points in the matrix are those for the temporal reference point *TR*, and for the start and end time-points of the task and its children. On this distance matrix, which corresponds to a subproblem P_τ of the global STN of the whole plan, we perform PC (line 4) and update the domains of the time-points in P_τ.

We then build a list L of *pending* children, whose sub-STN may need to be updated (line 5), and recurse to each child in this list (line 8). Note the list of pending children is initially set to all children of the root task. In making the recursive step, the local domain of the child in its distance matrix is the intersection of its local domain in A and its global domain. This ensures that all inference on the child's duration to date is propagated.

Once the recursive steps are all complete, if the local or global domain of any task in P_τ were updated as a result, we update the distance matrix and perform PC again (line 11);

[1] This, the standard semantics for task durations (Frank & Jónsson 2004; Wilkins 1999), means that, given two of the task's start, duration and end, we can compute bounds for the third.

this and subsequent invocations of PC may be incremental. Any child whose local domain changes as a result may have an impact on its siblings. Thus we must recurse again to all such children: these children are added to the new list L (line 15) and the loop repeats. The parent–children cycle terminates when parent and all child PC invocations are quiescent (line 16). If at any point a PC invocation finds an inconsistency, SR-PC halts and reports that the whole plan is temporally infeasible.

Worked Example

Figure 1 depicts a small HTN plan. The top-level objective task **A** has been decomposed (during the first expansion: first level to the second) into tasks **B** and **E**. **B** has been decomposed into **C** and **D**; and **E** into **F**–**H** (with two further expansions: second level to the third). The temporal bounds on each task are shown on the three timelines. Each task has start and end time-points. The constraints are the implied HTN constraints (not shown), some pairwise task ordering relations (depicted by $<$), and some quantitative STN constraints (depicted by the arrows).

The distance matrix of the global STN is shown in Figure 2; '`-`' denotes no explicit constraint between the two time-points, i.e. an uninformative $X_j - X_i \leq \infty$. Constraints prohibited by the sibling-restricted condition are shown by '`.`'; note the marked block structure of these entries. Recall that time-point domains are found in the first row and column; local domains of tasks are found in the off-diagonal entries. Although the matrix as shown is sparse, it will become dense after temporal propagation is complete.

On the STN this matrix represents, SR-PC considers subproblems with *TR* and the time-points of tasks as follows:

```
1:    TR, A
2:       TR, A, B, E
3:          TR, B, C, D
4:             C (leaf: just return)
5:             D (leaf: just return)
6:          TR, E, F, G, H
7:             F (leaf: just return)
8:             G (leaf: just return)
9:             H (leaf: just return)
10:      TR, A, B, E (no change)
11:   TR, A (no change)
```

To illustrate the propagation steps of SR-PC, consider its invocation with **A** as the root, i.e. line 2 (`TR, A, B, E`) above. After the initial call to PC, we recurse to each child: first to **B** (the next three lines), then to **E** (the following four lines). Since **A**'s distance matrix was updated by changed domains for **B** and **E** both, we call PC again (line 10 above, line 11 in Algorithm 1). After this step, neither child of **A** has had its global or local domain updated; thus there are no tasks in L for the next iteration (line 16 in Algorithm 1), and so SR-PC terminates.

Comparison with Naive PC By *naive PC* on a plan, we mean invoking PC on the distance matrix of the global STN: in the example, the matrix of Figure 2. We will assume, for n time-points, that an implementation of PC has time complexity $\Theta(n^3)$ for full propagation, and $\Theta(n^2)$ when incremental, i.e. for (re)computation for one time-point. This is

	TR	A		B		C		D		E		F		G		H	
TR	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
A	0	0	180	25	-	.	.	.	.	-	-	.	.	.	.	.	.
	-	0	0	-	0	.	.	.	.	-	0	.	.	.	.	.	.
B	-	0	-	0	80	0	-	-	-	-	-	-	-	-	-	-	-
	-	-	-	-20	0	-	0	15	0	-	-	-	-	-	-	-	-
C	-	.	.	0	-	0	-	-	-	.	.	.	.	.	.	.	.
	-	.	.	-	-	-5	0	-	-	.	.	.	.	.	.	.	.
D	-	.	.	0	15	-	0	0	10	.	.	.	.	.	.	.	.
	-	.	.	-	-	-	-	-5	0	.	.	.	.	.	.	.	.
E	-	0	-	-	0	.	.	.	.	0	-	0	-	-	-	-	-
	-	-	-	-	-	.	.	.	.	0	0	-	0	-	0	-	0
F	-	.	.	-	-	.	.	.	.	0	-	0	5	-	-	-	-
	-	.	.	-	-	.	.	.	.	-	-	0	0	-	-	15	-
G	-	.	.	-	-	.	.	.	.	0	-	-	0	0	-	-	-
	-	.	.	-	-	.	.	.	.	-	-	-	-	0	0	-	-
H	-	.	.	-	-	.	.	.	.	0	-	-	-	-	-	0	20
	-	.	.	-	-	.	.	.	.	-	-	-	-	-	-	-5	0

Figure 2: Complete distance matrix for the example HTN

fitting for a dense representation of distance matrices, as currently in PASSAT. Recall that each task in the plan has two time-points; the *TR* is an additional one time-point.

By counting operations, naive PC on Figure 1 has time 4913 (plus one distance matrix setup) and space 289. SR-PC has time 1758 (plus four matrix setups and two updates) and space 228 (plus stack space when navigating the tree of STNs). Thus, in this simple example, for approximately equal space, SR-PC reduces the time complexity by 67%.

Algorithm Properties

Because children can be added to the pending list (line 15) on every iteration, it is not obvious that the loop in the Algorithm 1 terminates. The proof comes from considering the circumstances when a local domain of a child task can be updated. We sketch the principle ideas, beginning with a necessary lemma.

Lemma 1. *Let τ be a task with no grandchildren, and A be its distance matrix formed by Algorithm 1 after PC has been initially applied (i.e. on first entry to the loop). Suppose the local domain of τ in A, d_τ, is tightened, and all other local domains held constant. When consistency is restored with PC, a further tightening of d_τ cannot occur.*

Proof. Suppose initially $d_\tau = [l, u]$. Since A is consistent, for every value $x \in [l, u]$ there exists a supporting tuple of values for the other time-points in A. Now let d_τ be tightened to $d'_\tau = [l', u'] \subseteq [l, u]$. Restore consistency of A with PC and suppose, for a contradiction, that PC narrows d'_τ further. This means there formerly existed an element $x' \in [l', u']$, now removed because it had no support. But $x' \in [l, u]$, contradicting the initial consistency of d_τ. □

The main result applies this lemma in structural induction over the tree of STNs considered by SR-PC. The base case is trivial, since Algorithm 1 returns immediately, with no changes to any domain, when invoked on a leaf node.

Theorem 2. *Let Π be a HTN plan with P its underlying (global) STN. Let τ_0 be the top-level objective task of Π. Algorithm 1 invoked on τ_0 terminates.*

Proof. Let $\lambda(\Pi)$ be the number of iterations through the loop in lines 6–16. We proceed by induction over the tree of STNs that SR-PC transverses. Consider a task τ with children $\tau_1, \ldots \tau_f$. Let A be the distance matrix created for τ. Observe that if a child has no children itself, then PC and so SR-PC invoked on the child's distance matrix affects no change (since all of the child's distance matrix is contained in A, and PC has been invoked on A in line 4). In particular, this holds when τ has no grandchildren.

On the first iteration through the loop, any of the local domains of τ or one or more τ_i may be tightened by the consistency restoration of A in line 11. As a result, any of the children may potentially be re-added to L.

On the second iteration through the loop, SR-PC is invoked on every child in L. Consider $\tau_i \in L$. The only change to τ_i's distance matrix A_i as formed by SR-PC on the recursive call, compared to A_i at the *end* of the previous call (i.e. during iteration 1), is that the local domain d_{τ_i} of τ_i may be narrowed. But by the inductive hypothesis and Lemma 1, then SR-PC affects no change to d_{τ_i}. Since this is true for every child $\tau_i \in L$, there are no changes to update to A when all the recursive calls have completed. Hence no children of τ are re-added to L on the second iteration, and since $L = \emptyset$ in line 16, the algorithm thus terminates with $\lambda(\Pi) \leq 2$. □

Theorem 2 tells us that, in the STN tree, we have $\lambda = 0$ for leaf nodes, 1 for nodes without grandchildren, and at most 2 otherwise.

With termination established, we can prove that the inference obtained by SR-PC is exactly the same as PC, i.e. Algorithm 1 is sound and complete.

Theorem 3. *Algorithm 1 invoked on τ_0 reports inconsistency iff P is inconsistent; otherwise it computes identical minimal domains as PC on P.*

Proof. Soundness is straight-forward, since SR-PC invokes PC on a set of submatrices of A_{τ_0}, where A_{τ_0} is the distance matrix of τ_0. Any inference it obtains is a subset of the inference produced by PC on A_{τ_0}.

Completeness is shown by examining the submatrices considered. Again, we use induction on the tree of sub-STNs. The sibling-restriction condition on temporal constraints means that A_{τ_0} contains (non-trivial) entries only for constraints between a parent task τ and its children τ_i, and between the children. All such constraints are considered by SR-PC. Moreover, on each pass through the loop, Algorithm 1 updates A_τ with any changes that can occur by PC from such constraints. Since the loop terminates only when these constraints imply no further change (or are found inconsistent), the domains resulting must be the same as that computed by PC on the distance matrix A_{τ_0}. □

Theoretical Complexity

For an HTN with approximately uniform branching from the top-level objective, we characterise an initial theoretical measure of expected time and space complexity of SR-PC. While recognising that many HTN plans are not uniform in this way, suppose a mean branching factor (i.e. number of children for each expanded task) f, and mean depth (i.e. number of expansions from root to primitive action) d. For example, Figure 1 has $f = 7/3$ and $d = 2$. Suppose a likelihood p that some child of a task updates a domain (lines 9 and 11 of Algorithm 1).

Our complexity analysis depends on $\lambda(\Pi)$, the number of iterations through the loop in lines 6–16. Theorem 2 proved $\lambda \leq 2$. The mean value of λ for a plan depends on p and on the entries of the distance matrices, i.e. on the temporal constraints. In practice, we find λ is often closer to 1 than 2. We are working to obtain a more precise characterisation by considering commonly-occurring structures of STN constraints.

At each level of the tree, if the current node is not a leaf, SR-PC creates a distance matrix with $1+2(1+f) = h$ time-points. It runs PC, then recurses on each child and possibly runs PC incrementally, repeating the last two steps λ times. Let $\text{SR}_{\text{time}}(d)$ be the average time complexity of SR-PC with depth d below the current node. Then we have:

$$\begin{aligned}
\text{SR}_{\text{time}}(d) &= \text{PC}_{\text{time}}(h) + \\
&\qquad \lambda\left(pf\text{PC}^{\text{incr}}_{\text{time}}(h,f) + f\text{SR}_{\text{time}}(d-1)\right) \\
&= (1+\lambda f)\left(\text{PC}_{\text{time}}(h) + \lambda pf\text{PC}^{\text{incr}}_{\text{time}}(h,f)\right) + \\
&\qquad (\lambda f)^2\text{SR}_{\text{time}}(d-2) \\
&= \cdots \\
&= \frac{(\lambda f)^d - 1}{\lambda f - 1}(\text{PC}_{\text{time}}(h) + \lambda pf\text{PC}^{\text{incr}}_{\text{time}}(h,f)) + 0
\end{aligned}$$

since $\text{SR}_{\text{time}}(0) = 0$ at a leaf node. $\text{PC}_{\text{time}}(k)$ denotes the complexity of PC on a matrix of size k, and $\text{PC}^{\text{incr}}_{\text{time}}(k,j)$ is its incremental complexity if j time-points are updated.

The total expected space, if the transversal of the tree is implemented depth-first, is $\text{PC}_{\text{space}}(dh)$. In comparison, naive PC creates a single distance matrix with $1+2fd$ time-points and runs PC on it.

Now recall that for dense matrices, $\text{PC}_{\text{space}}(k) = \Theta(k^2)$, $\text{PC}_{\text{time}}(k) = \Theta(k^3)$, and $\text{PC}^{\text{incr}}_{\text{time}}(k,j) = j\Theta(k^2)$. Expanding and dropping lower order terms gives expected time for SR-PC of order $\Theta(2\lambda pf^4\frac{(\lambda f)^d-1}{\lambda f-1})$, and for naive PC time $\Theta(8(df)^3)$. For both the expected space is $\Theta(4(df)^2)$.

Now since $\lambda \leq 2$ and $p \leq 1$, an upper bound on the time ratio of naive PC to SR-PC comes down to d^3 to $f(2f)^d$ when f is large. Being exponential compared to polynomial, SR-PC dominates as d increases. In practice, however, we find λ and f are small enough that d is never sufficiently large in actual plans for SR-PC to dominate. In this region $d \in [1, d^*]$ of practical interest, this theoretical comparison, albeit crude, in fact suggests SR-PC will have increasing advantage until d nears d^*.

Experimental Results

We have implemented SR-PC in the PASSAT system, with encouraging results. For an existing, real-world Special Operations Forces (SOF) domain,[2] Table 1 compares SR-

[2]The results are for a number of scenarios, including the hostage rescue scenario described in Myers *et al.* (2002).

plan	tasks, vars	d	f	SR-PC time	SR-PC space	SR-PC cpu	naive PC time	naive PC space	naive PC cpu	cpu ratio
airfield-1	40, 27	3.15	2.62	2200	169	0.01	531000	6560	0.49	49
airfield-2	108, 27	8.66	3.19	80600	6500	0.4	10200000	47100	5.21	13.02
recon-1	61, 15	3.73	1.96	31900	2980	0.24	1860000	15100	1.08	4.50
hostage-1	48, 16	3.66	3.91	23700	1910	0.07	913000	9400	0.62	8.86
hostage-2	59, 27	3.23	4.58	36900	2690	0.1	1690000	14200	1.092	10.92
hostage-3	169, 82	4.58	3.67	90800	6890	0.331	42500000	122000	21.73	65.65

Table 1: SR-PC and naive PC on SOF domain plans

PC and naive PC on a selection of complete and partial plans. Unless stated, the STN solver used for PC was PC-1 (Bessière 1996). The experiments were conducted on a 1.6GHz Pentium M with 512MB of memory, using Allegro Lisp 6.2.

The columns of Table 1 display, respectively, the name of the plan, the number of tasks and non-ground variables, the mean depth d and branching factor f of the HTN[3]; and for each method, measures of the number of operations for time and space (as in the earlier example), and the actual CPU runtime (in seconds). Note that all three measures are empirical: the time and space are counted operations during the experimental runs. The final column shows the ratio of CPU runtimes; greater than 1 is favourable to SR-PC. Overall, on these real plans, SR-PC outperforms PC by approximately an order of magnitude.

Figures 3, 4 and 5 present a comparison of SR-PC and naive PC on randomly generated plans from an abstract domain. The random generator accepts the parameters: minimum and maximum bounds on the depth d; the mean f and the maximum of a geometric distribution for the number of children of each node; and bounds on the number of temporal constraints in each expansion. The temporal constraints are chosen uniformly from a predefined set. The top-level objective was co-identified with *TR*, i.e. $X_{\tau_0} - TR = 0$.

Figure 3 shows counted operations for time and space, and the empirical runtime, as HTN depth d increases. The ratios between the two methods plotted are for $f = 1.4$; the y axis is shown with a log-scale. Note how the observed runtime ratio (denoted CPU) closely correlates with the time and space measures. Even for the plans of greatest depth, SR-PC performs propagation within user reaction time. For instance, when $d = 16$, SR-PC requires 0.21s compared to 73s for naive PC. Indeed, the runtime for SR-PC increases approximately linearly with d, while PC exhibits exponential growth. This behaviour is characteristic across other values of f as d varies.

Figure 4 shows the effect of varying the mean branching factor f. The ratios plotted are for $d = 5$; the y axis is a log-scale. Again we observe that the CPU runtime ratio lies between the time and space ratios. In contrast with the depth, as the branching factor increases, the advantage of SR-PC over PC begins to display a possibility of leveling off. Further experiments are needed to observe the trend at greater depths.

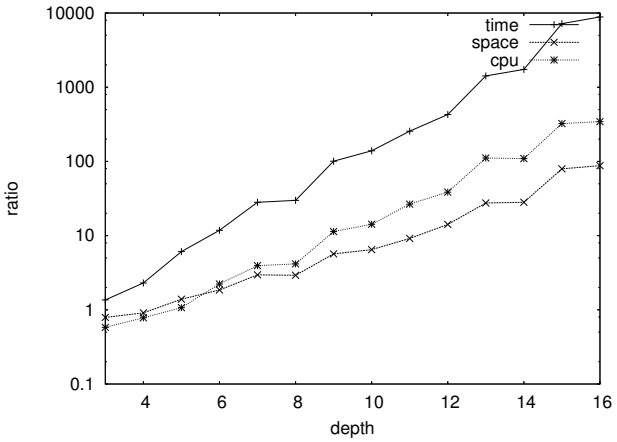


Figure 3: Mean time, space, and runtime ratios vs depth

Figure 5 shows the impact of the temporal consistency of the plan. The ratios are plotted as the probability that the global STN is consistent varies from 0 to 1. Note that no phase transition is observed. SR-PC has the greatest advantage when the probability of consistency is lower; for inconsistent problems, SR-PC is able to diagnose the inconsistency earlier. We conjecture this is because the cause of the inconsistency often arises from local interactions within a task network.

Discussion PASSAT is designed to assist the user in a mixed-initiative fashion. In such a user-interactive context, responsiveness of the system is crucial for effective plan authoring, even when developing significant plans with many temporal constraints. Thus, although the difference in absolute runtime for the SOF domains are in the order of seconds, temporal propagation with SR-PC makes the system noticeably and crucially more responsive.

The theoretical complexity of naive PC is cubic in the number of time-points. In practice as the plan size grows, our results suggest that the space required comes to dominate; this explains why PC exhibits exponential runtime growth in Figure 3. Our preliminary characterisation of the complexity of SR-PC in the last section indicates, in terms of f, a complexity of $\Theta(f^4 f^d)$ versus $\Theta(f^3)$ for PC. As Figure 4 suggests, this implies that the deeper the HTN tree

[3]Note that the complexity analysis of the last section considered an HTN tree with all leaves at depth d, rather than a general tree with mean depth d.

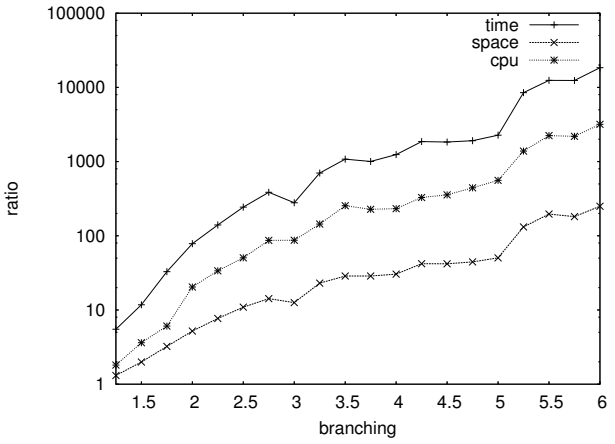


Figure 4: Mean time, space, and runtime ratios vs branching

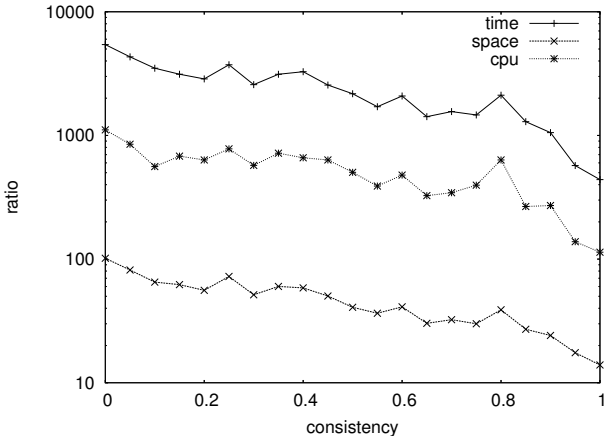


Figure 5: Time, space, and runtime ratios vs consistency

compared to its width, the greater the advantage (or the lesser the disadvantage, at least) of SR-PC if other factors are held constant. However, the influence of other factors is relevant, as shown by Table 1; consistency of the global STN is one of these.

By its operation, SR-PC automatically decomposes the global STN into sub-STNs based on the HTN structure. This is similar to decomposition of STN via its articulation points into biconnected components, which is known to be effective in speeding up propagation (Dechter, Meiri, & Pearl 1991). In our case, however, the global STN is a single biconnected component due to the implied HTN constraints. Thus SR-PC also propagates information between sub-STNs, via the parent task's distance matrix.

The STN solver is a black-box in SR-PC (in fact, within SR-PC, multiple methods for PC can be used on different occasions); the specific STN solver *S* parameterises SR-PC to the algorithm instance SR-*S*. Besides PC-1, we implemented the STN solver PC-2 in PASSAT, and compared PC-1, PC-2, SR-PC-1 and SR-PC-2. We found that building the queue of time-point triples in PC-2 quickly dominates the runtime, even for modest size plans. At least in our LISP implementation, PC-1 is much more practical than PC-2, and SR-PC-1 easily outperforms SR-PC-2.

As future work, the sophistication of △STP can be leveraged in SR-△STP (similar to how it can be leveraged as a black-box in a TCSP solver (Xu & Choueiry 2003)). For naive PC, △STP would be expected to outperform PC-1 because the initial distance matrices are relatively sparse — compare Figure 2. On the other hand, △STP is expected to bring a smaller benefit to SR-PC because the sub-STNs are smaller and more dense.

Conclusion and Future Work

We have presented an algorithm to efficiently perform temporal propagation on the Simple Temporal Network underlying a temporal hierarchical plan. Sibling-restricted propagation exploits the restricted constraints due to the HTN structure, decomposing the STN into a tree of sub-STNs. The SR-PC algorithm has been implemented in the PASSAT planning system, and empirical results demonstrate the effectiveness of the algorithm. Ongoing work is to provide a more precise theoretical characterisation of the average- and worst-case complexity.

While the results in PASSAT for SR-PC are favourable over naive PC, we have several improvements to make to the implementation. As noted, the present implementation uses PC-1 as the STN solver. Despite the small average size of the STNs solved by SR-PC, better performance is likely with a stronger solver, such as △STP. Second, there may be value in employing a sparse array representation. Third, coincidence of time-points is not actively exploited.

The reasoning problem addressed in this paper is determining the consistency and computing the minimal domains of time-points, for an STN underlying a plan. In both HTN and non-HTN planning, the plan is built incrementally; thus the associated STN is also built incrementally, and inference on it should exploit incremental constraint addition (and removal on backtracking). Incremental versions of classical STNs algorithms are widely used (Cesta & Oddi 1996). An important next step for us is therefore to extend SR-PC to an incremental version of the algorithm. In HTN planning, constraints are added (removed) when a task network is expanded (expansion backtracked). Besides making use of incremental STN solver, incremental SR-PC thus involves determining the highest task in the HTN tree that has changed, and considering the STN tree rooted at this task rather than at the top-level objective.

At present we are also implementing and experimenting with several orthogonal ideas to accelerate temporal propagation in an HTN planner. Unlike SR-PC, these ideas forgo completeness of the inference; in common with the method described here, they seek to exploit the HTN plan structure to do so in a principled way.

Acknowledgments Thanks to Hung Bui, Jeremy Frank and Karen Myers for helpful discussions on this topic, and to one of the reviewers in particular for pertinent questions.

References

Allen, J. F. 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM* 26(11):832–843.

Bessière, C. 1996. A simple way to improve path consistency in interval algebra networks. In *Proc. of AAAI-96*, 375–380.

Cesta, A., and Oddi, A. 1996. Gaining efficiency and flexibility in the simple temporal problem. In *Proc. of TIME-96*.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49(1–3):61–95.

Dechter, R. 2003. *Constraint Processing*. San Francisco, CA: Morgan Kaufmann.

Do, M. B., and Kambhampati, S. 2001. Planning as constraint satisfaction: solving the planning graph by compiling it into CSP. *Artificial Intelligence* 132(2):151–182.

El-Kholy, A., and Richards, B. 1996. Temporal and resource reasoning in planning: the parcPLAN approach. In *Proc. of ECAI-96*, 614–618.

Erol, K.; Hendler, J.; and Nau, D. 1994. Semantics for hierarchical task-network planning. Technical Report CS-TR-3239, Computer Science Department, University of Maryland.

Frank, J., and Jónsson, A. 2004. Constraint-based attribute and interval planning. *Constraints* 8(4):339–364.

Jonsson, A. K.; Morris, P. H.; Muscettola, N.; Rajan, K.; and Smith, B. 2000. Planning in interplanetary space: Theory and practice. In *Proc. of AIPS'00*, 177–186.

Laborie, P., and Ghallab, M. 1995. Planning with sharable resource constraints. In *Proc. of IJCAI'95*, 1643–1649.

Laborie, P. 2003. Algorithms for propagating resource constraints in ai planning and scheduling: existing approaches and new results. *Artificial Intelligence* 143(2):151–188.

Mali, A. D. 2002. Encoding temporal planning as csp. In *AIPS'02 Workshop on Planning for Temporal Domains*, 18–25.

Myers, K. L.; Tyson, W. M.; Wolverton, M. J.; Jarvis, P. A.; Lee, T. J.; and desJardins, M. 2002. PASSAT: A user-centric planning framework. In *Proc. of the Third Intl. NASA Workshop on Planning and Scheduling for Space*.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *J. Artificial Intelligence Research* 20:379–404.

Schwalb, E., and Vila, L. 1998. Temporal constraints: A survey. *Constraints* 3(2/3):129–149.

Smith, D.; Frank, J.; and Jónsson, A. 2000. Bridging the gap between planning and scheduling. *Knowledge Engineering Review* 15(1):47–83.

Son, T. C.; Baral, C.; and Tuan, L.-C. 2004. Adding time and intervals to procedural and hierarchical control specifications. In *AAAI-04*, 92–97.

Tate, A.; Drabble, B.; and Kirby, R. 1994. O-Plan2: An architecture for command, planning and control. In Fox, M., and Zweben, M., eds., *Intelligent Scheduling*. Morgan Kaufmann.

Tsamardinos, I.; Muscettola, N.; and Morris, P. H. 1998. Fast transformation of temporal plans for efficient execution. In *Proc. of AAAI-98*, 254–261.

Wilkins, D. E. 1999. *Using the SIPE-2 Planning System: A Manual for Version 6.1*. Artificial Intelligence Center, SRI International, Menlo Park, CA.

Xu, L., and Choueiry, B. Y. 2003. A new efficient algorithm for solving the simple temporal problem. In *TIME'03*, 212–222.

Yaman, F., and Nau, D. S. 2002. Timeline: An HTN planner that can reason about time. In *AIPS'02 Workshop on Planning for Temporal Domains*, 75–81.