COPLAS'07

International Joint CP/ICAPS Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems

Workshop Proceedings

Editors:

Miguel A. Salido, Antonio Garrido, Roman Bartak

Providence, Rhode Island, USA
September, 23, 2007
**Preface**

The areas of AI planning and scheduling have seen important advances thanks to the constraint satisfaction techniques. Now, many important real-world problems require dealing with efficient constraint techniques for planning, scheduling and resource allocation to competing goal activities over time in the presence of complex state-dependent constraints. Therefore, solutions to these problems must integrate resource allocation and plan synthesis capabilities. Basically, we need to manage complex problems where planning, scheduling and constraint satisfaction must be interrelated, which entail a great potential of application.

The workshop therefore aims at providing a platform for meeting and exchanging ideas and novel works in the field of AI planning, scheduling, constraint satisfaction techniques, and many other common areas that exist among them. In fact, most of the received works are based on combined approaches of constraint satisfaction for planning, scheduling and mixing planning and scheduling. The workshop will be held in September, 2007 in Providence (USA) as a joint workshop for both International Conference on Principles and Practice of Constraint Programming (CP'2007) and International Conference on Automated Planning & Scheduling (ICAPS'07).

All the submissions were reviewed by two anonymous referees from the program committee, who decided to accept 11 papers for oral presentation in the workshop. The papers provide a good mix of constraint satisfaction techniques for planning, scheduling, related topics and their applications to real-world problems. We hope that the ideas and approaches presented in the papers and presentations will lead to a valuable discussion and will inspire future research and developments in all the workshop participants.

The Organizing Committee.
September, 2007

Miguel A. Salido  
Antonio Garrido  
Roman Bartak
Organization

Organizing Committee

Miguel A. Salido, Universidad Politécnica de Valencia, Spain
Antonio Garrido, Universidad Politécnica de Valencia, Spain
Roman Bartak, Charles University, Czech Republic

Programme Committee

Federico Barber, Universidad Politécnica de Valencia, Spain
Roman Bartak, Charles University, Czech Republic
Chris Beck, University of Toronto, Canada
Daniel Borrajo, Universidad Carlos III, Spain
Luis Castillo, Universidad de Granada, Spain
Amedeo Cesta, ISTC-CNR, Italy
Minh Binh Do, Parc, USA
Antonio Garrido, Universidad Politécnica de Valencia, Spain
Hector Geffner, Universidad Pompeu Fabra, Spain
Enrico Giunchiglia, Università di Genova, Italy
Peter Jarvis, NASA Ames Research Center, USA
Inês Lynce, Universidade Técnica de Lisboa, Portugal
Derek Long, University of Strathclyde, UK
Pedro Meseguer, IIIA-CSIC, Spain
Alexander Nareyek, National University of Singapore, Singapore
Eva Onaindia, Universidad Politécnica de Valencia, Spain
Francesca Rossi, University of Padova, Italy
Miguel A. Salido, Universidad Politécnica de Valencia, Spain
Pascal Van Hentenryck, Brown University, USA
Gérard Verfaillie, ONERA, Centre de Toulouse, France
Vincent Vidal, CRIL-IUT, France
Dimitris Vrakas, Aristotle University of Thessaloniki, Greece
Toby Walsh, UNSW, Sydney and NICTA, Australia
Content

Application of Meta-Tree-Based Distributed Search to the Railway Scheduling Problem 6
Montserrat Abril, Miguel A. Salido, Federico Barber

An Efficient Model for Dynamic and Constrained Resource Allocation Problems 14
Camille Besse, B. Chaib-draa

Strong, Weak, and Dynamic Consistency in Fuzzy Conditional Temporal Problems 22
M. Falda, F. Rossi, K. B. Venable

Constraint Programming for Planning Routes in an E-learning Environment 30
Antonio Garrido, Eva Onaindia, Oscar Sapena

Extensions of the COMPLETION Constraint 38
András Kovács, J. Christopher Beck

Iterative Improvement Strategies for Multi-Capacity Scheduling Problems 45
Angelo Oddi, Amedeo Cesta, Nicola Policella, Stephen F. Smith

Planning and Scheduling Teams of Skilled Workers 52
Laurent Perron, Paul Shaw, Didier Vidal

Feasible Distributed CSP Models for Scheduling Problems 60
Miguel A. Salido

A Distributed CSP Approach for Solving Multi-agent Planning Problems 68
Oscar Sapena, Eva Onaindia, Antonio Garrido, Marlene Arangu

Plan Coordination for Durative Tasks 76
J. Renze Steenhuisen, Cees Witteveen

Constraint-Based Modelling of Discrete Event Dynamic Systems 84
Gérard Verfaillie, Cédric Pralet, Michel Lemaître

Author Index 92
Application of Meta-Tree-Based Distributed Search to the Railway Scheduling Problem*

Montserrat Abril, Miguel A. Salido, Federico Barber
Dpto. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia
Camino de Vera s/n, 46022, Valencia, Spain
{mabril, msalido, fbarber}@dsic.upv.es

Abstract

Many problems of theoretical and practical interest can be formulated as Constraint Satisfaction Problems (CSPs). Solving a general CSP is known to be NP-complete; however, distributed models may take advantage of dividing the problem into a set of simpler interconnected sub-problems which can be more easily solved. In this work, we present a distributed model for solving large-scale CSPs. Our technique carries out a partition over the constraint network by selection of tree structures; after partitioning, the sub-CSPs are arranged into a meta-tree CSP structure that is used as a hierarchy of communication by our distributed algorithm. We have focused our research on the railway scheduling problem which can be distributed by tree structures. We show that our distributed algorithm outperforms well-known centralized algorithms.

keywords: Distributed Constraint Satisfaction Problems, Tree Partition, Train Scheduling.

Introduction

Many real problems in Artificial Intelligence (AI) as well as in other areas of computer science and engineering can be efficiently modelled as Constraint Satisfaction Problems (CSPs) and solved using constraint programming techniques. Some examples of such problems include: spatial and temporal planning, qualitative and symbolic reasoning, diagnosis, decision support, scheduling, hardware design and verification, real-time systems and robot planning. Most of the work is focused on general methods for solving CSPs. However, many of the problems solved by using centralized algorithms are inherently distributed. Some works are currently based on distributed CSPs (see special issue of Artificial Intelligence, Volume 161, 2005).

Furthermore, many researchers are working on graph partitioning (Schloegel, Karypis, & Kumar 2003). The main objective of graph partitioning is to divide the graph into a set of regions such that each region has roughly the same number of nodes and the sum of all edges connecting different regions is minimized. Researchers are almost always interested in the size of nodes or edges, although a few studies have been made on the graph structure of the sub-problems induced by the partition (Miller 1986). For instance, one study seeks a node-separator whose induced graph is hamiltonian. Graph partitioning can also be applied to constraint satisfaction problems. Thus, we can use graph partitioning when dealing with large-scale CSPs to distribute the problem into a set of sub-CSPs. For instance, we can divide a CSP into several sub-CSPs so that constraints among variables of each sub-CSP are minimized (Salido & Barber 2006). Otherwise, a domain-dependent partition can be used (see Figure 7). This requires a deeper analysis of the problem to be solved. In this paper, we show that a domain-dependent partition obtains a more adequate distribution so that a higher efficiency is obtained.

In this paper, we present two techniques for structuring and solving binary CSPs. To this end, the binary CSP is previously organized into a meta-tree CSP structure so that the original constraint graph is partitioned into trees that represent sub-problems. Thus, the search algorithm carries out the search in each node in linear time (Freuder 1982), (Dechter & Pearl 1987).

Our research is also focused on the railway scheduling problem. Railway traffic has increased considerably, which has created the need to optimize the use of railway infrastructures. This is, however, a hard and difficult task. Our aim is to model the railway scheduling problem as a Constraint Satisfaction Problem (CSP) and solve it using a distributed CSP solver. Due to the topological properties of the railway scheduling problem, the resultant CSP can be distributed in semi-independent sub-problems so that the solution can be solved easier.

In the following section, we summarize some definitions about CSPs. A tree partition method is presented in section 3. Our distributed algorithm for solving meta-tree CSP structures is presented in section 4. In section 5 we describe the railway scheduling problem. An evaluation of our methods over real railway networks is presented in section 6. Finally, we summarize our conclusions in section 7.

*This work has been partially supported by the research projects TIN2004-06354-C02-01 (Min. de Educacion y Ciencia, Spain-FEDER), FOM-70022/IT05 (Min. de Fomento, Spain), GV/2007/274 (Generalidad Valenciana) and by the Future and Emerging Technologies Unit of EC (IST priority - 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

Copyright © 2007, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.
Centralized, Distributed and Partitionable CSPs

In this section, we present some basic definitions related to CSPs, which will be convenient for our purposes and will unify works from the constraint satisfaction community. Then, we present three ways for solving a CSP: as a centralized problem, as a partitionable problem and as a distributed problem.

A CSP consists of: a set of variables $X = \{x_1, x_2, ..., x_n\}$; each variable $x_i \in X$ has a set $D_i$ of possible values (its domain); a finite collection of constraints $C = \{c_1, c_2, ..., c_p\}$ that restricts the values that the variables can simultaneously take.

A solution to a CSP is an assignment of values to all the variables so that all constraints are satisfied; a problem with a solution is termed satisfiable or consistent.

A binary constraint network is a network in which every constraint subset involves at most two variables. In this case, the network can be associated with a constraint graph, where each node represents a variable and the arcs connect nodes whose variables are explicitly constrained (Dechter 1992).

A meta-tree CSP structure is a tree whose nodes are composed by trees. Thus, we will refer to the main tree as meta-tree CSP structure and to each individual tree as single-tree. We will define each node of the meta-tree CSP structure as meta-node and each individual and atomic node of the trees as single-node. It can be deduced that each meta-node corresponds to a single-tree. Each constraint between two single-nodes of different meta-nodes is called inter-constraint. Each constraint between two single-nodes of the same meta-node is called intra-constraint.

Partition: A partition of a set $C$ is a set of disjoint subsets of $C$ whose union is $C$. The subsets are called the blocks of the partition.

Distributed CSP: A distributed CSP (DCSP) is a CSP in which the variables and constraints are distributed among automated agents (Yokoo & Hirayama 2000).

Each agent has some variables and attempts to determine their values. However, there are inter-constraints and the value assignment must also satisfy them. In our model, there are $k$ agents $1, 2, ..., k$. Each agent knows the set of constraints and the domains of variables involved in these constraints.

Partition of CSPs

There are many ways to solve a CSP. However, these problems can be classified into three categories: centralized problems, distributed problems, and partitionable problems.

- A CSP is a centralized CSP when there are no privacy/security rules between parts of the problem, and all knowledge about the problem can be gathered into one process. It is commonly recognized that centralized CSPs must be solved by centralized CSP solvers. Many problems are represented as typical examples to be modelled as a centralized CSP and solved using constraint programming techniques. Some examples are: sudoku, n-queens, map coloring, etc.

- A CSP is a distributed CSP when the variables, domains and constraints of the underlying network are distributed among agents. This distribution is carried out due to many factors: constraints may be strategic information that should not be revealed to competitors, or even to a central authority; a failure of one agent can be less critical and other agents might be able to find a solution without the failed agent. Examples of such systems are sensor networks, meeting scheduling, web-based applications, etc.

- A CSP is a partitionable CSP when the global problem can be divided into smaller problems (sub-problems) which must be coordinated to find the solution to the global problem. For example, the search space of a CSP can be divided into several regions, and a solution is found by using parallel computing.

Given these three categories, we can conclude that a distributed CSP can not be solved by centralized techniques. However, can a centralized CSP be solved by distributed techniques? The answer is ‘yes’ if the CSP is divided previously.

Real problems usually imply models with a great number of variables and constraints, causing dense networks of inter-relations. Problems of this kind can be handled as a whole only at overwhelming computational cost. Thus, it could be an advantage to decompose problems of this kind into several simpler interconnected sub-problems which can be more easily solved.

In the following example, we show that a centralized CSP could be decomposed into several sub-problems in order to obtain simpler sub-CSPs. In this way, we can apply a distributed technique to solve the decomposed CSP.

The map coloring problem is a typically centralized problem. The goal of a map coloring problem is to color a map so that regions sharing a common border have different colors. Let’s suppose that each country of Europe must be colored. Figure 1 (1) shows a colored portion of Europe. This problem can be solved by a centralized CSP solver. However, if the problem is to color each region of each country of Europe (Spain, Figure 1(3); France, Figure 1(4)), it is easy to see that the problem can be partitioned into a set of sub-problems, grouped by clusters. This problem can be solved as a distributed problem, even when the problem is not inherently distributed.

A map coloring problem can be solved by first converting the map into a graph where each region is a vertex and an edge connects two vertices if and only if the corresponding regions share a border. In our problem of coloring the regions of each country of Europe, it can be observed that the corresponding graph maintains clusters representing each country (Spain, Figure 1(3); France, Figure 1(4)). Thus, the problem can be solved in a distributed way.

Why Tree Partition?

As Rina Dechter states in (Dechter 1992), a problem is considered easy when it admits a solution in polynomial time. In the context of constraint networks, a problem is easy if an algorithm like backtracking can solve it in a backtrack-free manner, i.e., without dead-ends, thus producing a so-
lution in linear time with regard to the number of variables and constraints. Theoretical researches has identified topological features that determine this level of consistency and has yielded tractable algorithms for transforming some networks into backtrack-free representations. The following paragraphs present a summary of this theory.

The theory is centered on a graphical parameter called width, and the definitions are relative to the primal constraint graph. An ordered (primal) constraint graph is defined as one in which the nodes are linearly ordered to reflect the sequence of variable assignments executed by the backtracking algorithm. The width of a node is the number of arcs that connect that node to previous ones, the width of an ordering is the maximum width of all nodes, and the width of a graph is the minimum width of all orderings of that graph. It is known that only trees are width-one graphs (Freuder 1982). An ordered constraint graph is backtrack-free if the level of directional strong consistency along this order is greater than the width of the ordered graph. Thus, if the graph has width-one (i.e., it is a tree), a directional two-consistency is sufficient (Dechter 1992) to solve the problem in linear time.

### How to Convert a binary CSP into a Meta-Tree CSP Structure

Any binary CSP can be translated into a meta-tree CSP structure. However, there are many ways to divide a graph into trees. Depending on user requirements, it may be desirable to obtain balanced single-trees, that is, each single-tree maintains roughly the same number of single-nodes; or it may be desirable to obtain single-trees in such a way that the number of edges connecting two single-trees are minimized.

Due to the complexity of finding the best partition, our technique finds an unbalanced tree partition in polynomial time \(n^2\) in the worst case, where \(n\) is the number of variables. It divides the problem into an undefined number of trees.

The TreePartition algorithm (Algorithm 1) divides the network graph \(G\) into \(k\) sub-graphs which are trees. The nodes and edges of graph \(G\) are, respectively, the variables and constraints of the CSP. TreePartition randomly selects a root node that does not belong to another sub-graph, then the SearchTree function constructs a tree. This function recursively carries out a Depth First Search in graph \(G\). The SearchTree function selects a new node \(i\) which is connected with \(\text{VarNode}\) and whose inclusion in the present \(\text{Tree}\) does not introduce a cycle, that is, node \(i\) is not connected with another node of the present \(\text{Tree}\). Node \(i\) is marked as visited; it indicates that this node already belongs to one tree. Tree construction finishes when either there are no unvisited nodes or the remainder nodes introduce cycles in the present tree. The TreePartition algorithm finishes when all nodes of \(G\) belong to any sub-graph.

#### Algorithm 1: TreePartition Algorithm

The next step is to build the meta-tree CSP structure with \(k\) meta-nodes that will be studied by agents. This meta-tree CSP structure is used as a hierarchy to communicate messages between meta-nodes. The meta-tree CSP structure is built using Algorithm 2. The nodes and edges of graph \(G\) are, respectively, the meta-nodes and inter-constraints obtained after the CSP partition. The root meta-node is obtained by selecting the most constrained meta-node. The MetaTreeCSPStructure algorithm then simply puts meta-node \(v\) into the meta-tree CSP structure \(\text{process}(v)\); it initializes a set of markers to indicate which vertices have

---

**Algorithm TreePartition**

**Input:** Graph \(G\), originally all nodes are unvisited.

Start meta-node \(v\) of \(G\)

**Output:** Tree\_partition

```
Tree\_partition=Ø;
while \(G\neq Ø\) do
    Tree=Ø;
    RootNode = selectNode(\(G\));
    insert RootNode into Tree;
    mark RootNode as visited;
    SearchTree(\(G\), RootNode, Tree);
    insert Tree into Tree\_partition;
end
end
```

**function SearchTree**\((G, \text{VarNode}, \text{Tree})\)

forall \(i\) adjacent\(^1\) to \(\text{VarNode}\) \& \(i\) unvisited do
    if NoCycle\((i, \text{Tree})\) then
        insert \(i\) into \(\text{Tree}\);
        mark \(i\) as visited;
        SearchTree\((G, i, \text{Tree})\);
    end
end

\(^1\) two single-nodes are adjacent if at least one constraint exists between them.

---

**Algorithm 2:** MetaTreeCSPStructure Algorithm

The MetaTreeCSPStructure algorithm then simply puts meta-node \(v\) into the meta-tree CSP structure \(\text{process}(v)\); it initializes a set of markers to indicate which vertices have
been visited; it chooses a new meta-node \( i \) and then calls \( \text{MetaTreeCSPStructure}(G, i) \) recursively. If a meta-node has several adjacent meta-nodes, it would be equally correct to choose them in any order, but it is very important to delay the test for whether a meta-node is visited until the recursive calls for previous meta-nodes are finished.

### Algorithm 2: MetaTreeCSPStructure Algorithm

Our aim is to solve CSPs by dividing the constraint graph by means of trees. Figure 3-1 shows a simple example of CSP. In Figure 3-2, this CSP has been divided into several trees and has been translated into a meta-tree CSP structure.

#### DFSTreeSearch Algorithm (DTS)

Our algorithm, called DFSTreeSearch (DTS), can be considered as a distributed and asynchronous algorithm. In the specialized literature, there are many works about distributed CSPs. In (Yokoo & Hirayama 2000), Yokoo et al. present a formalization and algorithms for solving distributed CSPs. These algorithms can be classified as centralized methods, synchronous backtracking or asynchronous backtracking (Yokoo & Hirayama 2000).

DTS is committed to solving the meta-tree CSP structure in a Depth-First Search Tree (DFS Tree) where the root meta-node is composed of the most constrained single-tree (in the sense that this single-tree maintains a higher number of single-nodes). DFS trees have already been investigated as a means to boost search (Decher 2003). Due to the relative independence of nodes lying in different branches of the DFS tree, it is possible to perform search in parallel on these independent branches.

Once the variables are divided and arranged, the problem can be considered as a distributed CSP, where a group of agents manages each single-tree with its variables (single-nodes) and its constraints (intra-constraints). Each agent is in charge of solving its own single-tree by means of the tree-solving algorithm defined in (Decher 2003). Each sub-problem is composed of its CSP subject to the variable assignment generated by the ancestor agents in the meta-tree CSP structure.

Thus, the root agent works on its sub-problem (root meta-node). If the root agent finds a solution, then it sends the consistent partial solution to its children agents in the meta-tree CSP structure, and all children work concurrently to solve their specific sub-problems knowing the consistent partial states assigned by the root agent. When a child agent finds a consistent partial state, it again sends this partial state to its children and so on. Finally, leaf agents try to find a solution to their own sub-problems. If each leaf agent finds a consistent partial state, it sends an OK message to its parent agent. When all leaf agents answer with OK messages to their parents, a solution to the entire problem is found. When a child agent does not find a solution, it sends a Nogood message to its parent agent. The Nogood message contains the variables that empty the variable domains of the child agent. When the parent agent receives a Nogood message, it stops the search of the children and tries to find a new solution taking into account the Nogood information and so on. If a parent agent finds a new solution, it will start the same process again sending this new solution to its children. Each agent works in the same way with its children in the meta-tree CSP structure. However, if the root agent does not find a solution, then DTS returns no solution found.

Figure 2 shows our technique for partitioning a CSP into a meta-tree CSP structure. Then, DTS is carried out. Once the CSP is partitioned, the root agent \((a_1)\) starts the search process finding a partial solution. It sends this partial solution to its children. Agents that are brothers are committed to concurrently finding the partial solutions of their sub-problems. Each agent sends the partial problem solutions to its children agents. A problem solution is found when all leaf agents find their partial solution. For example, \((\text{state } s_{12} + s_{43}) + (\text{state } s_{12} + s_{23} + s_{34})\) is a problem solution. The concurrence can be seen in Figure 2 in Time step 4 in which agents \(a_2\) and \(a_3\) are concurrently working. agent \(a_4\) sends a Nogood message to its parent (agent \(a_1\)) in step 9 because it does not find a partial solution. Then, agent \(a_5\) stops the search process of all its children and it finds a new partial solution which is sent to its children. Now, agent \(a_3\) finds its partial solution, and agent \(a_2\) works with its child, agent \(a_3\), to find their partial problem solution. When agent \(a_3\) finds its partial solution, the global problem will be found.
Example

Figure 3 shows an example to analyze the behavior of DTS.

First the constraint network of Figure 3(1) is partitioned into three trees and the DFS tree is built (Figure 3(2)). Agent $a$ finds its first partial solution $(X_1 = 1, X_2 = 1)$ and sends it to its children: agents $b$ and $c$ (see Figure 3(3)). This is a good partial solution for agent $c$ (Figure 3(4)); however this partial solution empties the $X_3$ variable domain. Thus, agent $b$ sends a Nogood message to its father (Nogood $(X_1 = 1)$) (Figure 3(5)). Then, agent $a$ processes the Nogood message, prunes its search space, finds a new partial solution $(X_1 = 2, X_2 = 2)$ and sends it to its children (Figure 3(6)). At this point in the process, agent $c$ sends a Nogood message to its father (Nogood $(X_1 = 2)$) because $X_5$ variable domain is empty (Figure 3(7)). Agent $a$ stops the search of agent $b$ (Figure 3(8)) and then processes the Nogood message, prunes its search space, finds a new partial solution $(X_1 = 3, X_2 = 3)$ and sends it to its children (Figure 3(9)). Since this last partial solution is good for both children, they respond with an OK message and the search ends (Figure 3(10)), returning the solution presented in Figure 3(11).

Railway Scheduling Problem

Train timetabling is a difficult and time-consuming task, particularly in the case of real networks where the number of constraints and the complexity of constraints grow drastically. A feasible train timetable should specify the departure and arrival time of each train to each location of its journey, in such a way that the line capacity and other operational constraints are taken into account. Traditionally, train timetables are generated manually by drawing trains on the time-distance graph called a running-map. The train schedule is generated from a given starting time and is manually adjusted so that all constraints are met. High priority trains are usually placed first followed by lower priority trains. It can take many days to develop train timetables for a line, and the process usually stops once a feasible timetable has been found. The resulting plan of this procedure may be far from optimal.

A sample of a running map is shown in Figure 4, where several train crossings can be observed. A running map contains information regarding railway topology (stations, tracks, distances between stations, traffic control features, etc.) and the schedules of the trains that use this topology (arrival and departure times of trains at each station, frequency, stops, crossings, etc.). The names of the stations are presented on the left side of Figure 4, and the vertical line represents the number of tracks between stations (one-way or two-way). The horizontal line represents the time.
the job-shop scheduling problem ([Silva de Oliveira 2001], (Walker & Ryan 2005)), where train trips are considered jobs that are scheduled on tracks that are regarded as resources.

Our goal is to model the railway scheduling problem as a Constraint Satisfaction Problem (CSP) and to solve it using constraint programming techniques. However, due to the huge number of variables and constraints that this problem generates, a distributed model is developed to distribute the resultant CSP into semi-independent sub-problems so that the solution can be found efficiently.

Variables and Constraints in the Railway Scheduling Problem

The variables of the railway scheduling problem are the arrival and departure times of trains at stations. The variables domain is the time with a granularity of minutes. There are three groups of scheduling rules in our railway scheduling problem: traffic rules, user requirements rules and topological rules. A valid running map must satisfy the above rules. These scheduling rules can be modelled using the following constraints, where variable $TA_i,k$ represents that train $i$ arrives at station $k$ and the variable $TD_i,k$ means that train $i$ departs from station $k$:

1. **Traffic rules** guarantee crossing and overtaking operations. We assume two trains ($i$ and $j$) going in opposite directions between stations $k$ and $k + 1$. The main constraints to take into account are:
   - **Reception time constraint.** There exists a given time to detour a train back from the main track so that crossing or overtaking can be performed (RecT).
     \[ (TA_{i,k} + RecT_i < TA_{j,k}) \lor (TA_{j,k} + RecT_j < TA_{i,k}) \]
   - **Expedition time constraint.** There exists a given time to put a train back on the main track so that crossing or overtaking can be performed (ExpT).
     \[ (TD_{i,k} + ExpT_i < TD_{j,k}) \lor (TD_{j,k} + ExpT_j < TD_{i,k}) \]
   - **Crossing constraint:** Any two trains going in opposite directions must not simultaneously use the same one-way track.
     \[ (TD_{i,k} + TD_{j,k+1}) < TD_{i,k+1} \lor (TD_{j,k} + TD_{i,k+1}) < TD_{j,k+1} \]
   - **Overtaking constraint:** Two trains ($i$ and $s$) going at different speeds in the same direction can only overtake each other at stations.
     \[ (TD_{i,k} < TD_{s,k}) \lor (TD_{s,k} + TD_{i,k+1} < TD_{s,k} + TD_{i,k+1}) \]

2. **User Requirements:** The main constraints due to user requirements are:
   - **Type and Number of trains** going in each direction to be scheduled.
   - **Path of trains:** Locations used and **Stop time** for commercial purposes in each direction.
     \[ TD_{i,k} = TA_{i,k} + StopTime_{i,k} \]

- **Scheduling frequency.** Train departure must satisfy frequency requirements in both directions. It could be a fixed time (1) or a time interval ($Freq \pm \delta$) (2). Frequency is a very tight constraint and is only sometimes required.
  - (1) $TD_{i+1,k} = TD_{i,k} + Freq$
  - (2) $(TD_{i,k} + Freq - \delta) \leq TD_{i+1,k} \leq (TD_{i,k} + Freq + \delta)$

- **Departure interval** for the departure of the first trains going in both the up and down directions.
  \[ StartTime_i < TD_{i,1} < EndTime_i \]

3. **Railway infrastructure topology and type of trains** to be scheduled give rise to other constraints to be taken into account. Some of them are:
   - **Number of tracks in stations** (to perform technical and/or commercial operations) and the number of tracks between two locations (one-way or two-way).
   - **Time constraints,** between each two contiguous stations ($TA_{i,(k+1)}$).
     \[ TA_{i,(k+1)} - TD_{i,k} = TA_{i,(k+1)} \]

Figure 6 shows the set of variables of two trains going in opposite directions between stations $A$ and $E$. After studying the problem, we have detected an advantageous tree partition of the railway scheduling problem. In Figure 6, the edges among two variables represent time constraints ($TD_{i,k} - TA_{i,(k+1)}$) and stop time constraints ($TA_{i,k} - TD_{i,k}$) respectively; these constraints make up train paths and they could be private information of railway operators. Figure 7 shows a clear tree partition of the railway scheduling problem where each sub-CSP has all variables of only one train and the respective *intra*-constraints are time constraints and time stop constraints that are usually fixed by railway operators. The *inter*-constraints are the traffic rules that are usually controlled by infrastructure managers.

**Evaluation**

In this section, we carry out an evaluation between DTS and a centralized CSP solver. To this end, we have used a well-known centralized CSP solver called Forward Checking (FC). The classical binary version of FC ([Haralick & G. 1980]) has a prohibitive computational cost for the type of problems evaluated in this section (a simple railway scheduling problem with 4 trains could not be solved after 1 day of execution); that is why we use the full path consistency Forward Checking algorithm (FCPath). This algorithm per-

---

**Figure 5:** Constraints related to crossing and overtaking in stations

![Diagram showing constraints related to crossing and overtaking in stations](image-url)

---

1FCPath were obtained from Van Beek page. It can be found in: http://ai.uwaterloo.ca/~vanbeek/software/software.html
forms full path consistency on future variables whenever the current variable is instantiated.

This empirical evaluation of the railway scheduling problem was carried out over a real railway infrastructure that joins two important Spanish cities (La Coruna and Vigo). The journey between these two cities is currently divided by 40 stations. In our empirical evaluation, each set of random instances was defined by the 3-tuple \(< n; s; f >\), where \(n\) was the number of periodic trains in each direction, \(s\) the number of stations, and \(f\) the frequency (in minutes). The problems were randomly generated by modifying these parameters. Usually, increasing the number of trains involves a CSP with a greater number of variables; increasing the number of stations involves a CSP with a greater number of variables and a greater domain size; and decreasing the frequency involves increasing the problem tightness because the number of conflicts between trains is greater.

In this evaluation, we compare the running time and concurrent constraint checks (CCC) ((Meisels et al. 2002)) of DTS with a well-known centralized algorithm: FCPath. DTS is executed over two different tree partitions: random partition is obtained using the TreePartition algorithm (Algorithm 1), which is a general tree partition method; train partition is a domain-dependent partition, which has been illustrated in Figure 7. It must be taken into account that both types of partitions roughly generate the same number of sub-problems and inter-constraints.

Figures 8 and 9 show the behaviors of DTS and FCPath in several instances of \(n\) according to the tuple \(< n, 5, 60 >\). The number of trains \((n)\) was increased from 1 to 20 trains in each direction. It must be taken into account that both graphs maintain a \(\log_{10}\) scale. Figures 8 and 9 show that DTS outperforms the FCPath algorithm, both random partition and train partition, in all instances. Figure 8 shows that DTS with train partition always has smaller running times than DTS with random partition. However, Figure 9 shows that DTS with random partition sometimes has fewer CCC than DTS with train partition. The two last assertions seem to be contradictory, but the explanation is in Figure 10: DTS with train partition exchanges fewer messages than DTS with random partition; thus, train partition saves a lot of running time. This is due to train partition involves better agent coordination.
Figure 11 show the behaviors of DTS and FCP in several instances of $s$ according to the tuple $<4, s, 60>$, where the number of stations ($s$) was increased from 5 to 20. It can be observed that DTS with train partition has a better behavior than DTS with random partition and FCP. In Figure 12, we evaluate the behavior of the algorithms with different frequencies according to the tuple $<4, 10, f>$, where frequency was increased from 15 to 60 minutes. It can be observed that due to the problem instances maintain the same number of variables and domain size, DTS with train partition and FCPPath maintain homogeneous behaviors. In general, both graphs corroborate the good behavior of the DTS algorithm, particularly with train partition.

Conclusions

We have presented two techniques for structuring and solving binary CSPs. The first one translates, in polynomial time, the original binary graph into a meta-tree CSP structure, where each node in the meta-tree CSP structure is a tree. The second technique is a distributed algorithm (DTS) for solving the resultant meta-tree CSP structure. DTS exploits the linear complexity to solve each tree and minimizes the storage of Nogoods. These techniques have been applied to the railway scheduling problem. The evaluation shows that general distributed models have a better behavior than the centralized model, but domain-dependent distributed models are more efficient than general ones. Thus, this technique may be appropriate for solving centralized problems that can be divided in smaller sub-problems.

References


Abstract
Dynamic constraint satisfaction is a useful tool for representing and solving sequential decision problems with complete knowledge in dynamic world and particularly constrained resource allocation problems. However, when resources are unreliable, this framework becomes limited due to the stochastic outcomes of the assignments chosen. On the contrary, Markov Decision Processes (MDPs) handle stochastic outcomes of unreliable actions, but their complexity explodes when using state-defined constraints. We thus propose an extension of the MDP framework so as to represent constrained and stochastic actions in sequential decision making. The basis of this extension consists in modeling the evolution of a dynamic constraint network by a MDP. We first study the complexity of the problem of finding an optimal policy for this model and then we propose an algorithm for solving it. Comparison to standard MDP shows that this framework noticeably improves policy computation.

Introduction
An autonomous agent, dynamically allocating stochastic resources to incoming tasks, faces increasingly complex situations when formulating its control policy. These situations are often constrained by limited resources of the agent, time limits, physical constraints or other agents. All these hindrances explain why complexity and state space dimension increase exponentially in size of the considered problem. Unfortunately, models that already exist either consider the sequential aspect of the environment, or its stochastic one or its constrained one. To the best of our knowledge, frameworks that model two of these aspects are not numerous, and even less numerous are frameworks that consider all three.

For example, dynamic constraint satisfaction problems (DCSPs) have been introduced by Dechter & Dechter (1988) to address problems that involve dynamics in constrained problems. In DCSPs, there is typically no transition model, and thus no concept of sequence of controls. Their objective is to minimize the work needed to repair a solution when a change occurs or to find robust solutions which could face changes. However, they are not in general used to plan in a stochastic world were solution at one step influences solutions in further steps when such knowledge is available.

On the other hand, Fargier, Lang, & Schiex (1996) proposed mixed CSPs (MCSPs) and probabilistic CSPs (PCSPs) (Fargier et al., 1997; Shazeer, Littman, & Keim, 1999), in which some uncontrollable variables model uncertainty. Solutions then depend on their possible values. While this model produces robust solutions, it does not deal with sequence of events or with unexpected events in which case a practical solver will have to be able to fall back to existing DCSP methods. Thus, this approach considers only the stochastic and the constrained aspects of the environment.

In their approach, Fowler & Brown (2003) included sequential decision making in their Branching Constraint Satisfaction Problem (BCSP) to model problems in which there is uncertainty in the number of variables. They construct a tree of the sequential assignment of variables depending on probability that changes income during resolution. This model includes stochastic, constrained and dynamic aspects, but does not deal with sequence of complete assignments and changes in variable set and/or constraint set as the DCSP framework does.

In the same context, Walsh (2002) proposed the Stochastic CSP (SCSP) framework that encompasses all models described above. Thus, his approach includes stochastic and constrained aspects as well as sequential decision making. However, this approach does not aim to choose the best sequence of assignments but only a sequence that is satisfied with a certain probability threshold. Indeed, there is no concept of valuation of an assignment as usually deal resource allocation problems.

Doigov & Durfee (2005a,b) presented several approaches to represent constraints in Markov Decision Processes mainly using linear programming techniques and weakly-coupled MDPs. These approaches are very efficient but suffer of limitation to linear constraints of linear programming and restriction to weakly-coupled agents.

In this paper, we introduce a new model based on DCSPs and Markov decision processes to address con-
strained stochastic resource allocation (SRA) problems by using expressiveness and powerfulness of CSPs. We thus propose a framework which aims to model dynamic and stochastic environments for constrained resources allocation decisions. Then, we present some efficient algorithms based on a combination of last researches in both constraint satisfaction and Markov decision problems. A complexity study is also made before presenting comparative experiments. We finally conclude with some remarks on the reasons for our model’s success and future work.

Background

A classical constraint satisfaction problem (CSP) is a triple \( P = (X, D, C) \), where \( X \) is the set of variables, each of them can take its possible values in a domain \( D \) (supposed here finite), and \( C \) is a set of constraints restricting the possible values of some variables. We will denote by \( S(P) \subset D^{n(X)} \) the set of all assignments satisfying all constraints \( C \) of \( P \). Solving a CSP is equivalent to finding one element \( \sigma \in S(P) \).

Dynamic Constraint Satisfaction Problem (DCSP)

According to Verfaillie & Jussien (2005) dynamic constraint satisfaction problem (DCSP) is a sequence of \( \eta \) CSPs \( \Psi = \{ P_1, ..., P_\eta \} \), each \( P_i \) depending only on changes in the definition of the previous one \( P_{i-1} \). These changes may affect any component in the problem definition: the set \( X \) of variables (by adding new variables or removing a subset of \( X \)), the set \( C \) of constraints (by adding, removing or modifying) and eventually domains \( D \) if they are not modifiable by changing variables or constraints. Formally:

**Definition 1. (Dynamic CSP)**

A dynamic CSP is a finite set \( \Psi = \{ P_i \} \) with each \( P_i = (X_i, D_i, C_i) \) where:

- \( X_i \) is the set of variables at step \( i \);
- \( D_i \) is the set of domains of these variables at step \( i \);
- \( C_i \) is the set of constraints restricting variables at step \( i \).

Solving a DCSP consists in finding a satisfying solution \( \sigma_i \in S(P_i) \) for each \( i, 1 \leq i \leq \eta \). Most past work on DCSPs has been devoted to finding a solution \( \sigma_{i+1} \) based on \( \sigma_i \). For instance, solution reuse and reasoning reuse (Verfaillie & Jussien, 2005) are two approaches that benefit from previous solutions to produce future ones. However, none of these approaches attempt to formalize the evolution of changes so they are not able to predict and control them.

Hence, dynamic CSPs are inadequate to address theory of control problems. We thus present Markov decision processes which are a well known approach in this domain before proposing a new framework based on the composition of these two approaches.

Markov Decision Processes (MDP)

A Markov Decision Process (MDP) models a planning problem in which action outcomes are stochastic but the world state is fully observable. The agent is assumed to know a probability distribution for action outcomes. Formally:

**Definition 2. (MDP)**

A MDP is defined by a tuple \( \mathcal{M} = (S, A, T, R, s_0) \) where:

- \( S \) is a finite set of states;
- \( A \) is a finite set of actions;
- \( T : S \times A \times S \rightarrow [0,1] \) is a probability distribution over \( S \) for any \( s \in S \) and \( a \in A \);
- \( R : S \times A \rightarrow \mathbb{R} \) is a bounded reward function;
- \( s_0 \) is the initial state;

Intuitively, \( T_{ss'}^a = Pr(s' | s, a) \) denotes the probability of moving to state \( s' \) when action \( a \) is performed at state \( s \), while \( R_{ss'}^a \) denotes the immediate utility associated with the action \( a \) in state \( s \).

Thus, solving an MDP aims to find an optimal policy \( \pi^* : S \rightarrow A \) that associates to each state \( s \in S \) an action \( a \in A \). \( \pi^* \) aims to maximize the expected reward accumulated:

\[
\pi^* = \sup_{\pi \in \Pi} \sum_{s \in S} V_\pi(s)
\]

where \( \Pi \) is the set of all policies.

Papadimitriou & Tsiklis (1987) have shown that finding an optimal policy as described above is one of the most difficult polynomial problems under some restrictions recalled in the section about complexity. Nevertheless, as entries can have an exponential size, this problem is one of the current major difficulties in the planning community. In fact, two main factors are responsible for this tractability problem: the exponential size of the state space and the branching factor between each state which make search barely feasible.

Many methods already exist that address the state space problem. For example, Dearden’s aggregation (Dearden & Boutilier, 1997) aims to group similar states while Sutton, Precup, & Singh (1999) organize states into a hierarchy to facilitate learning and planifi-

COPLAS 2007

15
see in the following sections, using constraints in factored MDPs in the context of resource allocation problems reduces also the state space. In fact, one can also imagine constraints that prune value of state variables that are not consistent with the considered problem.

**Markovian Constraint Satisfaction Problem (MaCSP)**

A Markovian CSP (MaCSP) is a Markov Decision Process which describes the stochastic evolution of a bounded dynamic constraint network. States represent possible configurations of the DCSP among its evolution, and actions represent assignments of each configuration of this DCSP. Thus, in a Markovian CSP, the Markov property is satisfied as a future configuration depends only on the previous configuration and the assignment chosen. Formally:

**Definition 3. (Markovian CSP)**

A Markovian CSP (MaCSP) is defined by a tuple \( \Phi = (S, \Psi, \{A\}, \mathcal{T}, \mathcal{R}, s_0) \) where:

- \( S = \{s_1, P_{s_1}\} \) where \( s_i \) is the state and each \( P_{s_i} = (X_{s_i}, D_{s_i}, C_{s_i}) \) is the current state of the underlying DCSP \( \Psi \) in the state \( s_i \);
- \( A_i = \{\sigma_i = \{x_1, \ldots, x_k\} : x_k \in D_{s_i}, \sigma_i \in S(P_{s_i})\} \) is the set of consistent assignments of variables in each state \( s_i, 1 \leq i \leq \eta \);
- \( T_{s_i, s_{i+1}}^\sigma = Pr(s_{i+1}|s_i, \sigma) \) is the transition probability that the assignment \( \sigma \) leads from a state \( s_i \) to a state \( s_{i+1} \);
- \( R^\sigma_s \) is a reward function leading to a satisfying subset of \( S \);
- \( s_0 \) is the initial state.

In other words, \( S \) models a factored state space where constraints \( C_{s_i} \) can exist between variables in each state \( s_i \). Variables are partitioned in two types: decision variables that are controllable and state variables that are not. \( A \) is the set of all possible assignments of decision variables, and \( A_i \) the subset of \( A \) that is consistent with current constraints in \( C_{s_i} \). The transition function \( T_{s_i, s_{i+1}}^\sigma \) represents the evolution of all variables over time (including state variables). Thus, if some constraints in \( C_{s_i} \) depend on state variables, then, these constraints may change over time (and in some cases appear or disappear). The reward function \( R^\sigma_s \) that assigns a value to each assignment of variables depending on the state \( s_i \) can be viewed in the resource allocation context as a prioritization of certain tasks over others or as the return given by achieving some tasks.

In fact, a DCSP is a particular case of MaCSP where the transition model \( T_{s_i, s_{i+1}}^\sigma \) is not specified and only decision variable exists. A DCSP can then be naturally defined as a MaCSP in which the transition model is deterministic whatever assignment chosen and reward function is \{satisfied, unsatisfied\}.

Moreover, as MaCSPs can be seen as factored MDPs as described by Boutilier, Dearden, & Goldszmidt (2000) where some constraints exist between variables of the factored MDP, this other point of view leads us to reconsider the transition function definition. In fact, as defined above, the transition function is a huge table which describes the evolution of each variable and of each constraint from one state to another. The size of the table is obviously non polynomial in the number of variables. So, an alternative way to represent this function is to describe evolution and dependencies of variables between different states or inside a same state by a Dynamic Bayesian Network (DBN) (Dean & Kanazawa, 1990) as described by Guestrin, Koller, & Parr (2001). Boutilier, Dearden, & Goldszmidt (2000) also noticed that this representation is at worst as compact as the explicit table and exponentially better in memory space in most of cases.

Solving a MaCSP consists in finding a consistent assignment \( \sigma_s \) for each state \( s_i \) defining a policy \( \xi : S \rightarrow A \) that associates for each state \( s_i \in S \) an assignment \( \sigma_s \in S(P_{s_i}) \) over the finite horizon \( \eta \). \( \xi^* \) is the optimal policy over the set of all policies \( \Xi \) such as:

\[
\xi^* = \sup_{\xi \in \Xi} V(\xi) = \sup_{\xi \in \Xi} \left[ \sum_{\sigma \in A} \sum_{s' \in S} \xi(\sigma, s') \mathcal{T}_{ss'}^\sigma R_s^\sigma + \gamma V(\xi(s')) \right]
\]

where \( 0 < \gamma < 1 \) is a discount factor.

**Complexity**

The complexity class of satisfying a dynamic CSP is easy to show since a dynamic CSP is a linear combination of CSPs in terms of complexity: As stated by Haralick et al. (1978), the problem of satisfiability of a constraint network as defined in previous section is NP-complete. Thus,

**Lemma 1.** The static unrestricted Constraint Satisfaction Problem is NP-complete.

**Theorem 1.** The Dynamic Constraint Satisfaction Problem is NP-complete.

**Proof.** First we show that the dynamic constraint satisfaction problem is solvable in polynomial time by a non-deterministic Turing machine. This is immediate, since the Turing machine can non-deterministically choose an assignment at each step and then verify its global consistency by checking each allowed tuple of each relation. Since \( R \), the factor size of constraints\(^1\) is fixed, the number of tuple to be checked is \( |\mathcal{X}| R \leq |\mathcal{X}|^R \) which is polynomial for each step and so for the \( \eta \) steps.

Now we show that some NP-complete problem would be solved by the dynamic constraint satisfaction problem. The problem to be considered is the static constraint satisfaction problem whose completeness is given by lemma 1. A CSP can be represented as a DCSP which has the same variables set and whose constraints

\(^1\)i.e. the maximum number of variables implied in each constraint, e.g. \( R = 2 \) for binary constraints.
are added sequentially to obtain the targeted CSP. Thus, solving this DCSP will solve the corresponding CSP. This transformation is clearly polynomial.

Furthermore, Papadimitriou & Tsikiklis (1987) have shown the following:

**Lemma 2.** (Papadimitriou & Tsikiklis, 1987) Given an MDP $\mathcal{M}$, a horizon $T$, and an integer $K$, the problem of computing a policy of a CSP in $\mathcal{M}$ under horizon $T$ that yields total reward at least $K$ is $P$-complete.

**Proof.** The same methodology is applied as for proposition 1. We first show that MaCSPs are solvable in polynomial time by a non-deterministic Turing machine. Since the Turing machine can non-deterministically choose an assignment in each state of the underlying MDP and then verify its consistency by checking each allowed tuple of each relation of the current CSP. Thus the number of tuples to be checked is still polynomial in $|\mathcal{X}| \times |\mathcal{S}|$.

To show that solving a MaCSP helps to solve another NP-complete problem, one can show that solving an MaCSP is equivalent to solve a DCSP where changes between two CSP are the same whatever assignment chosen and transition probability is one.

In fact we assume that, as in Markov Decision Processes, state space, action space and transition function are given and then we will just have to verify if actions are consistent with constraints which is linear in size of $\mathcal{A}$. Unfortunately, the practical problem associated to it is not as simple as stated above and we can conjecture that solving this problem is harder than it seems to be. In fact, we have to find all the solutions of a CSP in each state of the MDP to get the action space and this is already a $\#P$-complete problem. However, we intend that this framework aims to reduce branching factor by action pruning, but do not aim to be applied to huge instances of dynamic CSPs.

**Algorithms Combination**

As MaCSPs mixes a MDP and a CSP, two approaches for which different algorithms have been proposed, we have elaborated an algorithm (as depicted in Alg. 1) which uses efficient existing techniques developed in those contexts. Precisely, we chose a real-time dynamic programming (RTDP) approach for the Markovian aspect of MaCSPs since it is considered as the most effective both in time and in quality. In fact, RTDP-class algorithms are shown to converge to the optimal policy. We found in the literature several derivative versions of RTDP algorithms. We chose Focused RTDP (FRTDP)

---

**Algorithm 1** Focused RTDP for MaCSPs

```plaintext
1: function INITNODE(s):
2: //Implicitly called the first time each state s is touched
3: (s.L, s.U) ← (H_L, H_U); s.prio ← Δ(s)
4: end function

5: function FRTDP(s_0, ε, H_L, H_U, D_0, k_D):
6: D ← D_0
7: while s_0.U − s_0.L > ε do
8: (q_p, n_p, q_c, n_c) ← (0, 0, 0)
9: TRIALRECURSE(s_0, W = 1, d = 0)
10: if (q_c/n_c) ≥ (q_p/n_p) then D ← k_DD
11: end while
12: end function

13: function TRIALRECURSE(s, W, d):
14: (a^*, s^*, δ) ← backup(s)
15: TRACKUPDATEQUALITY(δW, d)
16: if Δ(s) ≤ 0 or d ≥ D then return
17: TRIALRECURSE(s^*, γT^ a^*, s^*, W, d + 1)
18: backup(s)
19: end function

20: function TRACKUPDATEQUALITY(q, d):
21: if d > D/k_D then
22: (q_c, n_c) ← (q_c + q, n_c + 1)
23: else (q_p, n_p) ← (q_p + q, n_p + 1)
24: end function

25: function BACKUP(s_i):
26: A_i ← BUCKETELIMINATION(s_i)
27: s_i,L ← max QL(s_i, a)
28: u ← max QU(s_i, a)
29: a^* ← sup QU(s_i, a)
30: δ ← |s_i,U − u|
31: s_i,U ← u
32: p ← max s_i+1 ∈ S γT^ a^*_{s_i,s_i+1}s_{i+1}.prio
33: s^* ← sup s_i,s_i+1 ∈ S γT^ a^*_{s_i,s_i+1}s_{i+1}.prio
34: s_i.prio ← min(Δ(s_i), p)
35: return (a^*, s^*, δ)
36: end function

37: function Δ(s):
38: return |s.U − s.L| − ε/2
39: end function

40: function QL(s, a):
41: return R(s, a) + γ ∑ s' ∈ S γT^ a_{s,s'}s'.L
42: end function

43: function QU(s, a):
44: return R(s, a) + γ ∑ s' ∈ S γT^ a_{s,s'}s'.U
45: end function
```
developed by Smith & Simmons (2006) which is currently, according to the authors, the most efficient, and which principally uses two bounds. The use of a lower bound offers guarantees in terms of value, and the upper bound guides efficiently the search. Moreover, CSP’s objective functions easily model lower bounds in planning problems. Thus, those kinds of algorithms can be also improved by CSP methods.

Concerning the constrained aspect, we use a bucket elimination procedure (line 26) that allows reducing the branching factor by selecting only feasible actions in a given state. Bucket elimination is essentially a method that propagates constraints applied on a variable of a CSP to the domain of the other variables involved in these constraints, and then eliminates the variable. Once all but one variables have been eliminated, a backward propagation on eliminated variables gives the set of all solutions. For details on bucket elimination, refer to (Dechter, 2003, chap. 13.3.3).

As in all RTDP-class, FRTDP’s execution consists in trials (line 17) that begin in a given initial state \( s_0 \) and then explore reachable states of the state space, selecting actions according to an upper bound. Once a final state is reached, it performs backups (line 18) on the way back to \( s_0 \). Backups consist essentially in Bellman updates which are applied on each bound (line 27, 28), on the priority of the state (line 32) and on the quality of the trial: the larger is the update on the upper bound, the better is the quality (line 30). This function returns the currently optimal action based on the priority and the quality of the backup.

As previously stated, FRTDP maintains a lower bound and uses a priority criterion (line 32) to select actions outcomes and to detect trial termination. The lower bound is used to establish the policy and it also contributes in the priority calculation of states to expand on the fringe of the search tree (line 34). Trial termination detection has been modified and improved by adding an adaptive maximum depth \( D \) (line 16) in the search tree in order to avoid over-committing to long trials early on. Indeed, the maximum depth \( D \) is lengthened (line 22) each time the trial is not useful enough. This usefulness is represented by \( \delta W \) where \( \delta \) measures how much the update changed the upper bound value of \( s \) and \( W \) the expected amount of time the current policy spends in \( s \), adding up all possible paths from \( s_0 \) to \( s \). Refer to the pseudo-code of Alg. 1 and to Smith & Simmons’ paper (Smith & Simmons, 2006) for details.

**Experimental Results**

A typical class of problems that can be solved by MaCSP are the dynamic, constrained and unreliable resource allocation problems. A number of different tasks must be achieved and actions consist in assigning various resources at different times to each of these tasks. Moreover, there are constraints on the resources, like time constraints, limited amount and interactions between those resources. Furthermore, these resources are unreliable in the sense that some of them are more effective when assigned to specific tasks and consequently probabilities are attached to their capacity to achieve tasks. In this case, the objective is to maximize over the temporal horizon the probability to achieve each task while respecting given constraints.

MaCSP is particularly suited for this problem since it involves stochasticity in actions that correspond to assignment of resources to tasks. Moreover, constraints on resources can easily be modeled in this framework using CSP. In a classical MDP, all constraints must be hard coded in states and hence will increase the state space while preventing constraints from evolving along time line. MaCSPs offer both the control on events via Markov processes and expressiveness via CSPs without exploding complexity (see theorem 2).

A typical example of this class of problems is the Dynamic Weapon-Target Allocation (DWTA) problem described by Hosein, Athans, & Walton (1988) which has been shown to be \( \mathbb{NP} \)-complete (Lloyd & Witsenhausen, 1986) even in the static unconstrained case.

The DWTA problem on which we experiment our algorithm is described as follows: Consider a naval platform attacked by a set \( \mathcal{M} \) of \( m \) threats. This platform owns a set \( \mathcal{W} \) of \( w \) weapons with limited ammunitions to defend itself on a finite temporal horizon \( T \). Thus its objective is to survive the attack by destroying threats with its weapons. Unfortunately, weapons are unreliable and weapon \( w_i \) have a probability \( p_{ij}^t \) to destroy threat \( m_i \) at time \( t \in \{0,\ldots,T\} \). As a result, the problem can be formally described by:

\[
\text{Maximize : } \sum_{t=1}^{T} \sum_{i=1}^{m} \left[ 1 - \prod_{j=1}^{w} (1 - \alpha_{ij}^t p_{ij}^t) \right]
\]

under \( \alpha_{ij}^t \in \{0,1\} \) and \( \sum_{i=1}^{m} \sum_{j=1}^{w} \alpha_{ij}^t = 1 \)

\( t \in \{1,\ldots,T\} \)

where \( \alpha_{ij}^t = 1 \) represents the assignation of weapon \( w_i \) on threat \( m_j \). The solution thus consists in a sequence of assignments of \( \alpha_{ij}^t \) for each \( t \in \{1,\ldots,T\} \).

Table 1: Examples of DWTA constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_1 )</td>
<td>A weapon must “see” the target (cf. table 2)</td>
</tr>
<tr>
<td>( C_2 )</td>
<td>A ML must be guided by a STIR from fire time to interception time, A Gun must use a STIR at fire time,</td>
</tr>
<tr>
<td>( C_3 )</td>
<td>Two STIRs cannot target the same threat.</td>
</tr>
</tbody>
</table>

Table 1: Examples of DWTA constraints

Moreover, the time for a weapon to intercept a threat depends on the range, the type and the speed of the threat and weapons also cannot freely fire at threats. Some constraints apply depending on their incoming azimuth and their distance from the platform. In our example, we consider that the platform is equipped with two
Separate & Track Illumination Radar (STIR), two Missile Launchers (ML), a mid-ranged Gun and a Close-In Weapon System (CIWS). To ease example understanding all threats are assumed to be the same type but with different starting range and speed\(^2\). The table 1 describes how weapons are constrained and table 3 what their probabilities of success are.

<table>
<thead>
<tr>
<th>Base State</th>
<th>0 to 360°</th>
<th>1 STIR, 1 Gun, 1 CIWS, 2 MLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sector</td>
<td>Angles</td>
<td>Difference from base state</td>
</tr>
<tr>
<td>A</td>
<td>350 to 10°</td>
<td>No CIWS</td>
</tr>
<tr>
<td>B</td>
<td>10 to 60°</td>
<td>No difference – Base state</td>
</tr>
<tr>
<td>C</td>
<td>60 to 120°</td>
<td>An additional STIR</td>
</tr>
<tr>
<td>D</td>
<td>120 to 150°</td>
<td>No difference – Base state</td>
</tr>
<tr>
<td>E</td>
<td>150 to 210°</td>
<td>No Gun</td>
</tr>
<tr>
<td>F</td>
<td>210 to 240°</td>
<td>No difference – Base state</td>
</tr>
<tr>
<td>G</td>
<td>240 to 300°</td>
<td>An additional STIR</td>
</tr>
<tr>
<td>H</td>
<td>300 to 350°</td>
<td>No difference – Base state</td>
</tr>
</tbody>
</table>

Table 2: Examples of DWTA constraints: Blind zones

<table>
<thead>
<tr>
<th>Weapon</th>
<th>Range</th>
<th>Probability of success</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML</td>
<td>From 2.2 to 20km</td>
<td>95%</td>
</tr>
<tr>
<td>Gun</td>
<td>From 1.5 to 5km</td>
<td>50%</td>
</tr>
<tr>
<td>CIWS</td>
<td>From 0.2 to 2km</td>
<td>30%</td>
</tr>
</tbody>
</table>

Table 3: Examples of DWTA outcomes of actions

In this application, we chose weapons as variables of the underlying DCSP and threats as values since constraints are mainly between weapons. The constraint network is trivial but is complex enough to prune the search and offers a gain versus a classical Markov Decision Process. Figure 1 shows the initial CSP and how it may evolve during an episode: There exists unary constraints that specify which threats are reachable regarding their distance from platform and the range of weapon, and binary constraints that bind firing weapons to STIRs depending on threats STIRs can “see”.

Figure 1: Example of a plan and the DCSP evolution during an episode

We then compare our framework to a classical MDP by specifying that actions that are not feasible have a null probability of success (e.g. a ML which fire to a too far target will never succeed): results in figure ?? show that MaCSP reduces the number of backups made and thus the policy computational time. Results were obtained with a FRDTP algorithm with the worst lower and upper bounds\(^3\).

Tendency in figure 2 shows that MDP computation is exponential in number of actions and also in number of tasks as it could be forecast. As a consequence, MaCSP which prunes useless actions has an exponential gain over the classical MDP. In a same way, as the branching factor depends on the number of actions, the number of backups (Figure 3) that FRDTP needs to converge with bad heuristics is also exponentially reduced. Results for four tasks and over with Markov Decision Processes were not obtainable due to lack of memory space.

\(^2\)Threats speed is assume to remain constant all over an episode.

\(^3\)Null lower bound in every state except in goal states and the upper bound is computed as if actions were deterministic.
Discussion & Further Work

This paper proposed a useful framework which combines Markovian Decision Processes and Constraints Satisfaction Problems. We studied the complexity of this model, proposed a simple algorithm to solve it and gave an example of application. Objectives of this approach have been to model actions space reduction due to environmental constraints and then reduce the branch factor of the state space in order to facilitate the search of optimal policies with efficient algorithms like the Real-Time Dynamic Programming family (Bonet & Geffner, 2003; McMahan, Likhachev, & Gordon, 2005; Smith & Simmons, 2006).

Nonetheless, it has some drawbacks: the action space of a MaCSP with \( x \) constrained variables which have each a domain of size \( d \) is in \( O(x^d) \) which is obviously exponential in size of the domains. Thus, in the worst case, if there not enough constraints to prune the action space it will be as prejudicial to search all the solutions of this huge CSP, which is a \#P-complete problem, as searching directly a solution with an efficient algorithm (assuming all actions are feasible but have a null probability to lead to another state). However, in the average, a gain may be done.

Regarding further, apart from studying the efficiency of this framework in some others problems, there are mainly two future research avenues. First, factored MDP of Boutilier, Dearden, & Goldszmidt (2000) is a very well suited framework to apply MaCSP, with constraints between state variables, allowing also to prune non consistent states if a priori knowledge about environment is available. Some work on framework generalization has already been made by Pralet, Verfaillie, & Schiex (2006) where an algebraic framework called PFU (standing for Plausibility, Feasibility, Utility) that encompasses MDP, CSP and also Bayesian Networks and Influence Diagrams has been proposed. However this framework does not encompass MaCSP since our framework models the evolution of constraints by allowing constraints between decision variables and state variables. Studying the consequences of an extension of PFU in this way could thus be interesting.

The second interesting research avenue is about Graphical Games Theory proposed by Kearns, Littman, & Singh (2001). Constraint satisfaction algorithms were recently applied to solve graphical games by representing interactions between agents by constraints. Graphical Games could lead by the mean of this framework to stochastic graphical games with few efforts.

Nevertheless, some other aspects still have to be studied in MaCSPs such as optimality criteria, bounds on error made by adding constraints and existing algorithm convergence. This framework also opens many research avenues since it combines two well-known techniques where much work has been done in recent years. Learning and Reinforcement Learning is one example from the Markovian community; solution reuse and reasoning reuse is another example from the dynamic CSP community.

Acknowledgements

We would like to thank Pascal Tesson for his helpful comments on complexity results, Gérard Verfaillie and Cédric Pralet for their comments on the model and reviewers for their useful overall suggestions.

References


Strong, Weak, and Dynamic Consistency in Fuzzy Conditional Temporal Problems

M. Falda and F. Rossi and K. B. Venable
Dept. of Pure and Applied Mathematics
University of Padova, Italy

Abstract

Conditional Temporal Problems (CTPs) allow for the representation of temporal and conditional plans, dealing simultaneously with uncertainty and temporal constraints. In this paper, CTPs are generalized to CTPPs by adding preferences to the temporal constraints and by allowing fuzzy thresholds for the occurrence of some events. The usual consistency notions (strong, weak and dynamic) are then extended to encompass the new setting, and their corresponding testing algorithms are provided. We show that the complexity of the algorithms does not increase w.r.t. their classical counterparts for CTPs. We also show that our framework generalizes STPUs as well, another temporal framework with uncertainty and preferences. This means that controllability in STPPUs can be translated to consistency in CTPPs, indicating a strong theoretical connection among the two formalisms.

Introduction

Many systems and applications need to be able to reason with alternative situations, plans, contexts and to know what holds in each of them. Moreover, they may have to set temporal constraints on events and actions. Conditional Temporal Problems (CTPs) (Tsamardinos, Vidal, & Pollack 2003) are a formalism that allows for modeling conditional and temporal plans which deal with the uncertainty arising from the outcome of observations and with complex temporal constraints. In CTPs the usual notion of consistency is replaced by three notions, weak, strong and dynamic consistency, which differ on the assumptions made on the knowledge available.

Another class of temporal reasoning problems that deals with similar scenarios are Simple Temporal Problems with Uncertainty (STPUs) (Vidal & Fargier 1999). In such problems the uncertainty lies in the lack of control the agent has over the time at which some events occur. Such events are said to be controlled by “Nature”. In STPUs consistency is called controllability and, similarly to CTPs, there are three notions, weak, strong and dynamic controllability, based on different assumptions made on the uncontrollable variables. Despite the fact that consistency in CTPs and controllability in STPUs appear similar, their relation has not been formally investigated.

Furthermore, in rich application domains it is often necessary to handle not only temporal constraints and conditions, but also preferences over the execution of actions. Preferences have been added to STPUs in (Rossi, Venable, & Yorke-Smith 2006); in addition to expressing uncertainty, in STPPUs contingent constraints can be soft, meaning that different preference levels are associated to different durations of events.

In this paper we introduce the CTPP model, an extension of CTPs which adds preferences to the temporal constraints and generalizes the simple Boolean conditions to fuzzy rules; these rules activate the occurrence of some events on the basis of fuzzy thresholds. Moreover, also the activation of the events is characterized by a preference function over the domain of the event. This provides an additional gain in expressiveness, allowing one to model the dynamic aspect of preferences that change over time.

Quantitative temporal constraint problems have been used for many applications in practice, ranging from space applications (MAPGEN (Ai-Chang et al. 2004)) to temporal databases (Combi & Pozzi 2006) and personal assistance (Autominder, (Pollack et al. 2003)). We expect CTPPs to be useful in all of the above.

After defining CTPs with fuzzy preferences, we extend all the consistency notions of CTPs. Moreover, we provide algorithms for testing such new notions which are in the same complexity class as their classical counterparts. Finally, we show how the STPPUs are related to CTPPs by providing a mapping from STPPUs to CTPPs (and thus also from STPUs to CTPs) which preserves the controllability/consistency notions. In particular, such a mapping proves that CTPPs are a more expressive model. All proofs have been omitted for lack of space.

Background

STPs and STPPs. A Simple Temporal Problem (STP) (Dechter, Meiri, & Pearl 1991) is defined as a set of variables $V$, each of which corresponds to an instantaneous event, and a set $E$ of constraints between the variables. The constraints are binary and are of the form $l_{ij} \leq x_i - x_j \leq u_{ij}$ with $x_i, x_j \in V$ and $l_{ij}, u_{ij} \in \mathbb{R}$; $l_{ij}$ and $u_{ij}$ are called the bounds of the constraint.

Preferences have been introduced in STPs by (Khatib et al. 2001), defining Simple Temporal Problems with Pref-
ferences (STPPs). In particular, a soft temporal constraint \(<I,f>\) is specified by means of a preference function on the interval, \(f : I \rightarrow [0,1]\), where \(I = [l_1, u_1]\). An STPP is said to be consistent with preference degree \(\alpha\) if there exists an assignment of its variables that satisfies all constraints and that has preference \(\alpha\). The preference of an assignment is obtained by taking the minimum of the preferences given by each constraint to the projection of the assignment onto its variables. An optimal solution is one such that there is no other solution with higher preference. Such a solution can be found in polynomial time (Khatib et al. 2001).

**STPUs and STPPUs.** STPUs (Vidal & Fargier 1999) are STPs in which the temporal constraints are divided in two classes: those representing durations under the control of the agent (called requirement constraints) and those representing durations decided by “Nature” (called contingent constraints). Such a partition induces a similar partition over the variables. In (Rossi, Venable, & Yorke-Smith 2006) STPUs are extended to preferences by replacing STP constraints with soft temporal constraints. Thus an STPPU is a tuple \(<N_e,N_c,L_e,L_c>\) where \(N_e\) is the set of executable timepoints, \(N_c\) is the set of contingent timepoints, \(L_e\) is a set of soft requirement constraints, and \(L_c\) is a set of soft contingent constraints. The notions of controllability of STPUs are extended to handle preferences. Here we focus on two of such notions. An STPPU is said to be \(\alpha\)-strongly controllable if there is a fixed way to assign the values to the variables in \(N_e\) such that whatever Nature will choose for the variables in \(N_c\), the resulting assignment is either optimal (if Nature’s choice prevents from achieving preference level \(\alpha\)) or it has preference \(\alpha\). Optimal weak controllability simply requires the existence of an optimal way to assign values to the variables in \(N_e\) given any assignment to those in \(N_c\).

**CTPs.** CTPs (Tsamardinos, Vidal, & Pollack 2003) extend temporal constraint satisfaction problems (Dechter, Meiri, & Pearl 1991) by adding observation variables and by conditioning the occurrence of some events on the presence of some properties of the environment. A CTP is a tuple \(<V,E,L,OV,O,P>\) where \(P\) is a set of Boolean atomic propositions, \(V\) is a set of variables, \(E\) is a set of temporal constraints between pairs of variables in \(V\), \(L : V \rightarrow \mathcal{Q}^*\) is a function attaching conjunctions of literals in \(\mathcal{Q} = \{p_i : p_i \in P\} \cup \{\neg p_i : p_i \in P\}\) to each variable in \(V\). \(OV \subseteq V\) is the set of observation variables, and \(O : P \rightarrow OV\) is a bijective function that associates an observation variable to a proposition. The observation variable \(O(A)\) provides the truth value for \(A\). In \(V\) there is usually a variable denoting the origin time, set to 0. In this paper this variable will be denoted by \(x_0\). Thus, in CTPs, variables are labelled with conjunctions of literals, and the truth value of such labels are used to determine whether variables represent events that are part of the temporal problem. In this paper we consider only CTPs where \(E\) contains only STP constraints. In a CTP, for a variable to be executed, its associated label must be true. The truth values of the propositions appearing in the labels are provided when the corresponding observation variables are executed. The constraint graph of a CTP is a graph where nodes correspond to variables and edges to constraints. Nodes \(v\) is labeled with \(L(v)\) and edge \(e\) is labeled with the interval of constraint \(e\). Labels equal to true are not specified. An execution scenario \(s\) is a conjunction of literals that partitions the set of variables in two subsets: the subset of the variables that will be executed because their label is true given \(s\), and the subset of the other variables, that will not be executed. \(SC\) is the set of all scenarios. Given a scenario \(s\), its projection, \(Pr(s)\), is the set of variables that are executed under \(s\) and all the constraints between pairs of them. \(Pr(s)\) is a non-conditional temporal problem.

Figure 1 shows an example inspired from (Tsamardinos, Vidal, & Pollack 2003). The example is about a plan to go skiing at station \(Sk1\) or \(Sk2\), depending on the condition of road \(R\). Station \(Sk2\) can be reached in any case, while station \(Sk1\) can be reached only if road \(R\) is accessible. If \(Sk1\) is reachable, we choose to go there. Moreover, temporal constraints limit the arrival times at the skiing station. The condition of road \(R\) can be assessed when arriving at village \(W\). In the figure, variables \(XY_1\) and \(XY_2\) represent the start and the end time for the trip from \(X\) to \(Y\). Node \(O(A)\), where \(A = \text{“road } R\text{ is accessible”}\) is \(HW_r\). There are two scenarios, \(A\) on variables \(\{x_0, HW_s, HW_e, WSk1, WSk2\}\) and \(\neg A\) on variables \(\{x_0, HW_s, HW_e, WSk2, WSk1\}\).
Dynamic Consistency (DC). Dynamic consistency (DC) assumes that information about observations becomes known during execution. A CTP is dynamically consistent if it can be executed so that the current partial solution can be consistently extended independently of the upcoming observations.

The CTP depicted in Figure 1 is not DC. In fact, if A is true then we have to leave home after 10, if A is false we have to leave home before 8. However, being at village W is a precondition for the observation of proposition A and this fact prevents us to observe A before leaving home. Therefore, we cannot distinguish between the two scenarios A and ¬A in time to schedule our departure from home accordingly.

Fuzzifying CTPs

The conditional nature of CTPs is enclosed in the variables’ labels, whose truth value enables or disables the presence of variables in the problem. Such labels indeed act as rules that select different execution paths, which, given variable v and its label L(v), can be written as follows: IF L(v) THEN EXECUTE (v).

The idea of fuzzifying such kind of rules has been already taken into consideration, for example in the field of fuzzy control (Lee 1990; Cox 1992). In fact, real world objects often do not present a crisp membership and classical Logics has difficulties to describe some concepts (e.g., “tall”, “young”, etc.). Another problem is that temporal informatics has difficulties to describe some concepts (e.g., “tall”, “young”, etc.). Another problem is that temporal informatics has difficulties to describe some concepts (e.g., “tall”, “young”, etc.).

In a general study of such rules (Dubois & Prade 1996), both the premise and the consequence of the rule have been equipped with truth degrees associated with them. We will do the same for CTP’s rules.

In our case, however, these two degrees have different meanings: the degree of the premise is used to establish if the variable should be executed, and therefore provides a truth value; the degree of the consequence, instead, can be considered as a preference on the execution of the variable.

Boolean propositions were justified in CTPs, where labels were evaluated in a crisp way, but in CTPs they would reduce the expressiveness of the fuzzy rules; for this reason CTPPs will be equipped with a set P of fuzzy atomic propositions and a set of fuzzy literals \(Q = \{p_i : p_i \in P\} \cup \{-p_i : p_i \in P\}\) which are mapped to values from \([0, 1]\) by an interpretation function.

**Definition 1 (Interpretation function).** An interpretation function is a function \(deg : W \subseteq Q \rightarrow [0, 1]\), where \(l \in W\) iff \(\neg l \in W\) and \(vl \in W\), \(deg(\neg l) = 1 - deg(l)\).

The rules we will use to fuzzify CTPs are of the form

\[
\text{IF } p(t(L(v), deg) > \alpha \text{ THEN EXECUTE } (v) : cp(min(L(v), deg))}
\]

where \(L(v) \in Q^*\) is the “fuzzy” label of variable \(v\), \(deg\) is an interpretation function, function \(pt\) gives the truth degree of \(L(v)\) given \(deg\), and \(cp\) is the preference function associated with the consequence. The set of all “truth-preference” fuzzy rules will be named FR.

To interpret a conjunction of fuzzy literals, given an interpretation \(deg\), it is natural to take their minimum degree, as usual in conjunctive fuzzy reasoning. Thus function \(pt : Q^* \rightarrow [0, 1]\) will be the min operator.

**Definition 2 (pt function).** Let \(L(v) = \land_i l_i, v \in V, l_i \in W \subseteq Q\), and \(deg : W \rightarrow [0, 1]\), then \(pt(L(v), deg) = min(deg(l_1), \ldots, deg(l_n))\).

For example, a fuzzy proposition A representing sentence “It is hot” can be true with different degrees. We could say it is true with degree \(deg(A) = 0.4\) if the outside temperature is mild, and with degree \(deg(A) = 0.8\), if the outside temperature is above 80°F. Similarly a fuzzy proposition B representing sentence “I’m thirsty” can reasonably have different truth degrees. We can imagine attaching to a variable, representing the time at which we go buy a cold drink, label \(L(v) = AB\). This will allow us to construct a rule for \(v\) which will activate variable “get cold drink” only if the heat level or the thirst are above a given threshold.

Since we will always use the above function \(pt\), each rule can be characterized by its threshold and its preference function. Thus we will sometimes denote a rule via the notation \(r(\alpha, cp)\).

Each fuzzy rule states that variable \(v\) is part of the problem if value \(pt(L(v), deg)\) is greater than the threshold \(\alpha\). Moreover, the consequence specifies the preference associated with the execution of \(v\). In general, such a preference can depend on the truth degree of the premise and on the time at which \(v\) is executed. Therefore, it is reasonable to define \(cp : [0, 1] \rightarrow (\mathbb{R}^+ \rightarrow [0, 1])\), that is, as a function which takes in input the truth degree of the premise, i.e., \(pt(L(v), deg)\), and returns a function which, in turn, takes in input an execution time and returns a preference in \([0, 1]\).

In other words, function \(cp\) allows us to give a preference function on the execution time of \(v\) which depends on the truth degree of the label of \(v\). However, this also allows us to model situations where the preference function for the activation of \(v\) is independent of the truth degree of the premise, as a special case in which function \(cp\) has type \(cp : \mathbb{R}^+ \rightarrow [0, 1]\). This restricted kind of rules will be named r-cp.

In CTPs, a variable without a label implicitly has a label with value true. Similarly, in the fuzzy extension we consider, any variable whose associated rule is not specified has the following implicit one: IF true THEN EXECUTE (v) : 1. This means that variable \(v\) is always present in the problem, and its execution has preference 1 independently of the execution time.

**Definition 3 (CTPP).** A CTPP is a tuple \(<V, E, L, R, OV, O, P, >\) where:

- \(P\) is a finite set of fuzzy atomic propositions with truth degrees in \([0, 1]\);
- \(V\) is a set of variables;
- \(E\) is a set of soft temporal constraints between pairs of variables \(v_i, v_j\).
• \( L : V \rightarrow Q^* \) is a function attaching conjunctions of fuzzy literals \( Q = \{ p_i : p_i \in P \} \cup \{ \neg p_i : p_i \in P \} \) to each variable \( v_i \in V \);

• \( R : V \rightarrow \mathcal{F} \mathcal{R} \) is a function attaching a “truth-preference” fuzzy rule \( r(\alpha, cp) \) to each variable \( v_i \in V \);

• \( OV \subseteq V \) is the set of observation variables;

• \( O : \mathcal{P} \rightarrow OV \) is a bijective function that associates an observation variable to each fuzzy atomic proposition. Variable \( O(A) \) provides the truth degree for \( A \).

As explained above, the execution of a variable \( v \in V \) depends on the evaluation of the fuzzy rule associated with it. A value assigned to a variable \( v \in V \) represents the time at which the action represented by \( v \) is executed; this value will be also written as \( T(v) \). If \( v \) is an observation variable it also represents the time at which the truth degree of the observed proposition is revealed.

Once a CTPP is defined, it is advisable to check statically if the information on labels and rules is consistent similarly to what is done in CTPs. In particular, if a variable \( v \) is executed, all the observation variables of the propositions in its label \( L(v) \) must have been executed before \( v \). In CTPs this is tested by checking if for each \( v \in V \) and for each proposition \( A \in L(v), L(v) \supseteq L(O(A)) \) and \( T(O(A)) < T(v) \), where \( O(A) \) is the observation node of proposition \( A \).

In the fuzzy case, where conjunction is replaced by min and the truth values of the propositions are in \([0, 1]\), \( L(v) \supseteq L(O(A)) \) has to be augmented with the condition that the threshold in the rule associated with \( O(A) \) should not be lower than the threshold of the rule associated to \( v \). More formally:

**Definition 4 (Structural Consistency).** Let \( v \) be a variable of a CTPP and \( L(v) \) its label. A CTPP is **structurally consistent** if each observation variable, say \( O(A) \), which evaluates a fuzzy proposition \( A \in L(v) \), is such that \( L(O(A)) \subseteq L(v) \) and \( \alpha \geq \beta \), where \( R(v) = r(\alpha, cp) \) and \( R(O(A)) = r(\beta, cp') \).

Checking the structural consistency of a CTPP can be performed in \( O(|L(V)|^2) \) since to establish the consistency of the label of a variable at most \( O(|L(V)|) \) labels (and thresholds) must be considered.

The definitions of scenario, projection, schedule and strategy are analogous to the classical counterparts.

**Definition 5 (Scenario).** Given an CTPP \( P \) with a set of fuzzy literals \( Q \), a **scenario** is an interpretation function \( s : W \rightarrow [0, 1] \) where \( W \subseteq Q \) that partitions the variables of \( P \) in two sets: set \( V_1 \), containing the variables that will be executed and set \( V_2 \) containing the variables which will not be executed. A variable \( v \), with associated rule \( r(\alpha, cp) \), is in \( V_1 \) iff \( pt(L(v), s) \geq \alpha \), otherwise it is in \( V_2 \). \( S(P) \) is the set of all scenarios of \( P \).

**Definition 6 (Partial scenario).** A **partial scenario** is an interpretation function \( s : W \rightarrow [0, 1] \) where \( W \subseteq Q \) that partitions the variables of the CTPP in three sets: set \( V_1 \), containing the variables that will be executed, set \( V_2 \) containing the variables which will not be executed and set \( V_3 \) containing the variables the execution of which cannot be decided given the information provided by \( s \). A variable \( v \), with associated rule \( r(\alpha, cp) \) and label \( L(v) \), is in \( V_1 \) iff \( L(v) \supseteq W \), is in \( V_1 \) iff \( pt(L(v), s) \geq \alpha \), otherwise it is in \( V_2 \).

Since a scenario chooses a value for each fuzzy literal, it determines which variables are executed and also which preference function must be used for their execution. This means that a scenario projection must contain the executed variables, the temporal constraints among them, and the information given by the preference function of each of the executed variables. This information can be modelled by additional constraints between the origin of time and the executed variables.

**Definition 7 (Constraints induced by a scenario).** Given a (possibly partial) scenario \( s \) and a variable \( v \) executed in \( s \), consider its associated rule \( r(\alpha, f) = R(v) \). The constraint induced by this rule in scenario \( s \) is the soft temporal constraint \( cst_s(v) \) defined on variables \( x_0 \) and \( v \) by \((0 \leq v - x_0 < +\infty) \) with associated constraint preference function \( f(\min_{A \subset L(v)} f(A)) \). The constraints induced by scenario \( s \) are all the constraints induced by variables executed in \( s \), that is, \( U(s) = \{ cst_s(v, v \in s) \} \).

**Definition 8 (Scenario projection).** Given an CTPP \( P \) and a scenario (or partial scenario) \( s \) of \( P \), its **projection** \( Pr(s) \) is the STPP obtained by considering the set of variables of \( P \) executed under \( s \), all the constraints among them, and the constraints in \( U(s) \). Two scenarios are **equivalent** if they induce the same projection.

**Definition 9 (Schedule).** A **schedule** \( T : V \rightarrow \mathbb{R}^+ \) of a CTPP \( P \) is an assignment of execution times to the variables in \( V \). Given a scenario \( s \) and a schedule \( T \), the preference degree of \( T \) in \( s \) is \( pref_s(T) = \min_{c_{ij} \in Pr(s)} f_{ij}(T(v_j) - T(v_i)) \), where \( f_{ij} \) is the preference function of constraint \( c_{ij} \) defined over variables \( v_i \) and \( v_j \). We indicate with \( T \) the set of all schedules.

Given a CTPP \( P \) an execution strategy \( St : S(P) \rightarrow T \) is a function from scenarios to schedules.

Figure 2 shows an example of CTPP that extends the CTP in Figure 1. There are three skiing stations: \( Sk_1, Sk_2 \) and \( Sk_3 \). A represents the fuzzy proposition “there is no snow”; station \( Sk_1 \) is the least accessible, so it is reachable only if \( A \) is at least 0.8; on the other hand, station \( Sk_3 \) has the most reliable roads, so it is accessible when \( A \) is above 0.3; station \( Sk_2 \) has intermediate reachability conditions, so it is accessible for values of \( A \) above 0.5. At the same time, however, the higher the snow, the more preferable it is to go skiing. For this reason, the \( cp \) functions of the rules are “inversely” proportional to the truth degree of observation \( A \). For example, this function could be \( cp(x) = (1 - x) \). The two temporal constraints of the original example from \( x_0 \) to \( W Sk_1c \) and to \( W Sk_3c \) have been fuzzified by using trapezoidal preference functions. The preference functions for the other constraints have been omitted, meaning that they are constant functions always returning 1. In this example there are four distinct scenarios, given by \( s_1(A) = 1, s_2(A) = 0.8, s_3(A) = 0.5 \), and \( s_4(A) = 0.3 \). Thus projection \( Pr(s_1) \) is the STPP defined on variables \( x_0, HW_5, HW_6, W Sk_1, W Sk_1c \), projection \( Pr(s_2) \) is
the STPP over variables \( x_0, HW_s, HW_e, WSk2_s, WSk2_e \), projection \( Pr(s_{13}) \) is the CTPP over \( Sx_0, HW_s, HW_e, WSk3_s, WSk3_e \), and projection \( Pr(s_4) \) is the STPP over \( x_0, HW_s, HW_e \).

**Consistency notions in CTPPs**

Consistency notions in CTPPs are analogous to the ones in CTPs. However, we now have to consider also the preferences. There are again three notions of consistency depending on the assumptions made about the availability of the uncertain information.

**Definition 10 (\( \alpha \)-Strong Consistency).** A CTPP is \( \alpha \)-strongly consistent if there is a viable execution strategy \( St \) such that, for every scenarios \( s_1 \) and \( s_2 \), and variable \( v \) executed in both,

1. \([St(s_1)](v) = [St(s_2)](v)\);
2. the global preference of \( St(s_1) \) and of \( St(s_2) \) is at least \( \alpha \).

In words, to be \( \alpha \)-strong consistent, we must have a schedule that satisfies all the constraints independently of the observations, giving a global preference greater than or equal to \( \alpha \). This is the strongest consistency notion since it requires the existence of a single schedule that gives preference at least \( \alpha \) in every scenario. On the contrary, we can just require the existence for every scenario of a schedule (possibly a different one for different scenarios) that has a preference of at least \( \alpha \) given the corresponding projection. This notion is that of \( \alpha \)-weak consistency.

**Definition 11 (\( \alpha \)-Weak Consistency).** A CTPP \( Q \) is said \( \alpha \)-weakly consistent (\( \alpha \)-WC) if, for every scenario \( s \in S(Q) \), \( Pr(s) \) is consistent in the STPP sense with preference degree at least \( \alpha \).

The above definitions are at the two extremes w.r.t. assumptions made on which events will be executed: \( \alpha \)-SC assumes no knowledge at all, while \( \alpha \)-WC assumes the scenario is given. A notion consistency which lies in between is \( \alpha \)-dynamic consistency which assumes that the information on which variables are executed becomes available during execution in an on-line fashion. In order to define it, we first need to recall the concept of observation history from CTPs and say when a partial scenario and a scenario are consistent.

**Definition 12 (Observation History).** Given a scenario \( s \) and a schedule \( T \), for each variable \( v \) we define the observation history of \( v \) w.r.t. schedule \( T \) and scenario \( s \) as the set \( H(v, s, T) \) containing the observations performed before time \( T(v) \).

**Definition 13 (Cons(s,w)).** Given a CTPP \( P \) and scenario \( s \) we say a partial scenario \( w \) is consistent with \( s \), written \( Cons(s, w) \) if: STPP \( Pr(w) \) is a sub-problem of STPP \( Pr(s) \), in the sense that the set of variables (resp. constraints) of \( Pr(w) \) is a subset of the set of variables (resp. constraints) of \( Pr(s) \) and no variable executed given \( s \) is not executed given \( w \).

This last definition extends the one given in the classical case, where it is sufficient to say that a partial assignment is consistent with a scenario if the variables executed by the partial assignment are a subset of those executed by the scenario. We will use this notion in the definition of \( \alpha \)-Dynamic Consistency, to express when at a given time the set of observations collected at that time is consistent with a scenario.

**Definition 14 (\( \alpha \)-Dynamic Consistency).** A CTPP is said \( \alpha \)-dynamically consistent if there exists a viable execution strategy \( St \) such that \( \forall o \) and for each pair of scenarios \( s_1 \) and \( s_2 \) \([Cons(s_1, H(v, s_1, St(s_1))) \land Cons(s_1, H(v, s_2, St(s_2)))) \Rightarrow [St(s_1)](v) = [St(s_2)](v)\) and the global preferences of \( St(s_1) \) and \( St(s_2) \) are at least \( \alpha \).

In words, a CTPP is \( \alpha \)-DC if for every variable \( v \), whenever two scenarios \( s_1 \) and \( s_2 \) are not distinguishable at the execution time for \( v \) \([Cons(s_1, H(v, s_1, St(s_1))) \land Cons(s_1, H(v, s_2, St(s_2)))) \), there is an assignment to \( v \) \([St(s_1)](v) = [St(s_2)](v)\) which can be extended to a complete assignment which in both scenarios will have preference at least \( \alpha \).

It is easy to see that, as for CTPs, \( \alpha \)-SC \( \Rightarrow \alpha \)-DC \( \Rightarrow \alpha \)-WC. Moreover, given \( \alpha \in [0, 1] \), if a CTPP is \( \alpha \)-SC/DC/WC then it is \( \beta \)-SC/DC/WC \( \forall \beta \leq \alpha \).

In what follows we consider a property which is common to all three the consistency notions. In order to do so we consider a subclass of CTPPs characterized by a special type of truth-preference rules. We will then show that the consistency of general CTPPs is equivalent to the consistency of a related problem in such a subclass.

**CTPPs with restricted rules.** We start by considering a simplified case, that is, when the preference functions of the rules are independent of the truth degree of the label \( pt(L(v), deg) \). In such a case, given rule \( r_1(\alpha, f) \), we assume that \( f \) is an \( r_\text{-cp} \) function. CTPPs with such a restriction will be denoted by \( R \)-CTPPs.

The preference information given by \( f \) can be equivalently expressed by adding a constraint between the origin of time \( x_0 \) and the variable to which rule \( r \) is associated. More precisely, the constraint induced by \( f \) is the soft temporal constraint \( cst(v) \) defined on variables \( x_0 \) and \( v \) by \( 0 \leq v - x_0 < +\infty \) with associated preference function.
The constraints induced by a whole CTPP $Q$ are all the constraints induced by the variables of $Q$, that is, $U(Q) = \{\text{cst}(v), v \text{ variable of } Q\}$.

In the specific case of an R-CTPP $Q$, the preference function of each constraint in $U(Q)$ will just be $f(\alpha)$, since in this case $f$ does not depend on the truth value of the propositions in the premise of the rule.

**Theorem 1.** Given a CTPP $Q = < V, E, L, R, OV, O, P >$, let us define a function $R'$ from $R$ as follows: if $R(v) = r(\alpha, f)$, then $R'(v) = r(\alpha, f')$ where $f' = \min_{\beta \in [0,1]} f(\beta)$. Then $Q' = < V, E, L, R', OV, O, P >$ is an R-CTPP. Moreover, $Q$ is $\alpha$-SC/DC/WC if and only if $Q'$ is $\alpha$-SC/DC/WC.

**Testing consistency of CTPPs**

Thanks to Theorem 1, when testing the consistency of a CTPP we can restrict ourselves to testing the consistency of its related R-CTPP without loss of generality.

**Testing $\alpha$-SC**

The algorithm we propose to test the $\alpha$-SC of an R-CTPP is based on the correspondence of the $\alpha$-SC of the R-CTPP and the consistency preference degree of a related STPP.

**Theorem 2.** Given an R-CTPP $M = < V, E, L, R, OV, O, P >$, let $E' = E \cup U(M)$. Then $M$ is $\alpha$-strongly consistent if and only if the STPP $< V, E', >$ is consistent with preference degree $\alpha$.

Theorem 2 relates the $\alpha$-SC of an R-CTPP to the consistency level of an STPP. This allows us to check the $\alpha$-SC of an R-CTPP by just constructing the appropriate STPP and then finding its best consistency level. This will give us the highest level $\alpha$ at which the R-CTPP is $\alpha$-SC. Since, under some tractability assumptions, solving a fuzzy STPP can be done in polynomial time (Khatib et al. 2001), $U(Q)$ contains $O(|V|)$ constraints, the procedure takes polynomial time.

**Testing $\alpha$-Weak Consistency**

In classical CTPs, the problem of checking WC is a co-NP complete problem (Tsamardinos, Vidal, & Pollack 2003). Therefore, being CTPPs an extension of CTPs, we cannot expect to do better. The classical algorithm to test the WC of CTPs checks the consistency of all complete scenarios by identifying a set of labels $LS$ that covers all the scenarios (Tsamardinos 2001). As seen in the example in Figure 2, the scenarios of a CTPP are determined not only by the labels used in the problem, but also by the thresholds levels. However, in the case of R-CTPPs, the definition of equivalence between scenarios collapses to that for CTPs, that is, two scenarios are equivalent if they induce the same partition of the variables. In fact, in R-CTPPs the preference on the induced constraint is independent of the value of the observation in the head of the corresponding rule. Thus the projection of the scenario is fully specified by the set of executed variables.

We first define for each literal $l \in Q$ an auxiliary set $M(l)$ that contains the set of the threshold levels of truth-preference rules defined on labels containing $l$. More precisely: $M(l) = \{\alpha_i : \exists v \in V \text{ with } R(v) = r(\alpha_i, cp) \text{ and } l \in L(v)\} \cup \{1\}$.

Given set $M(l)$ for each literal $l$, we consider scenarios mapping each literal $l$ into a value in $M(l)$.

**Definition 15 (Meta-scenario).** Given a CTPP $P$ with set of fuzzy literals $Q$ a meta-scenario is an interpretation function $ms : (W \subseteq Q) \rightarrow \cup_{l \in W} M(l)$ such that $ms(l) \in M(l)$, $\forall l \in W$. We will denote the set of meta-scenarios as $MS(P) \subset S(P)$.

Given the equivalence relation defined on R-CTPP scenarios, every scenario $s \in S(P) \setminus MS(P)$ is equivalent to a meta-scenario $ms \in MS(P)$.

**Theorem 3.** Given an R-CTPP $P$, $\forall s \in S(P)$, $\exists ms \in MS(P)$ s.t. $Pr(s) = Pr(ms)$.

In particular, from the above theorem we can immediately deduce that a R-CTPP is $\alpha$-WC if and only if all projections of meta-scenarios are consistent with optimal preference level at least $\alpha$. However, two meta-scenarios in $MS(P)$ can be equivalent. In order to further reduce the set of projections to be considered, we apply a procedure similar to that proposed in (Tsamardinos, Vidal, & Pollack 2003), in order to find a minimal set of meta-scenarios containing only one meta-scenario for each equivalence class. We refer to this procedure as Algorithm FST.

Algorithm FST takes in input a set of propositions $SL$, a current partial meta-scenario $s$, the set $ExecVars$ of variables which can be executed given the information in $s$, the set $PV$ containing the sets of executed variables already considered, and, finally, the set $MS$ of meta-scenarios selected so far. In output, it gives set of meta-scenarios $MS'$. First considers if the set of propositions $SL$ is empty and, if so, it returns the current set of meta-scenarios $MS$. Otherwise, it chooses (in some pre-fixed order) proposition $H$ and then removes it from $SL$. Next, for each threshold $\alpha$ (in increasing order) in the set $M(H)$, it extends the current meta-scenario with assignment $H = \alpha$ and computes the set of variables $ExecVars$ which are or could be executed given the information in $s$. In more detail, procedure $ConsVars$ takes in input a set of variables $X$, a partial meta-scenario $w$, and a CTPP $P$, and returns the subset of variables of $X$ containing only variables that in $P$ are associated with a rule whose head is not false given $w$ (set $V_1 \cup V_3$ according to the notation of Definition 6).

If set $ExecVars$ has not been considered before (that is, it is not contained in set $PV$) then, if either all the propositions in $SL$ have been considered or $ExecVars$ is empty, then $ExecVars$ is added to set $PV$ and the set of meta-scenarios $MS$ is updated with the new meta-scenario found $s$. Otherwise, if neither of the above sets are empty the search is carried on recursively.

In order to find a minimal set of meta-scenarios of an R-CTPP $P$ with proposition set $P$. Algorithm FST is called with $SL = P$, $s = \text{nil}\ 1 ExecVars = V$, $PV = \emptyset$, $MS = \emptyset$.

---

1We write $s = \text{nil}$ meaning the function with the empty domain, that is, to model a partial scenario in which no proposition is assigned.
The key idea of the algorithm is that as we extend a partial scenario the set of variables that could be executed can only shrink. Moreover, since for each proposition \( H \) the thresholds in \( M(H) \) are considered in increasing order, when a set of executed variables is found, all its subsets have already been considered and thus if such a set is already in \( PV \) the search can avoid the recursive call.

**Theorem 4.** Consider an R-CTPP \( P \) with proposition set \( \mathcal{P} \). Let \( MS' \) be the set of meta-scenarios returned by Algorithm FST when called on \( SE = \mathcal{P} \), \( s = \text{nil} \), \( \text{ExecVars} = V \), \( PV = \emptyset \), \( MS = \emptyset \). Then:

- \( \forall s \in MS', s \in MS(P) \);
- \( \forall s', s \in MS(P), \exists s \in MS' \) such that \( Pr(s') = Pr(s) \);
- \( \forall s, s' \in MS', Pr(s) \neq Pr(s') \);

The complexity of Algorithm FST is \( O(\Pi_{H \in \text{metaScenarios}}|M(H)|) \) since, in the worst case the algorithm explores the whole set of meta-scenarios, of size \( \Pi_{H \in \text{SE}(\mathcal{P})}|M(H)| \).

**example**

Consider the following R-CTPP with four variables \( v_1, v_2, v_3, v_4 \) whose associated rules are \( R(v_1) = r(0.3, \text{IF } A > 0.3 \text{ THEN EXECUTE } v_1 : 1) \), \( R(v_2) = r(0.5, \text{IF } A > 0.5 \text{ THEN EXECUTE } v_2 : 1) \), \( R(v_3) = r(0.2, \text{IF } AB > 0.2 \text{ THEN EXECUTE } v_3 : 1) \), and \( R(v_4) = r(0.5, \text{IF } AB > 0.5 \text{ THEN EXECUTE } v_4 : 1) \).

In this case \( M(A) = \{0.2, 0.3, 0.5, 1\} \) and \( M(B) = \{0.2, 0.5, 1\} \). This problem has 12 meta-scenarios, while the minimal set is \( \{\{A = 0.2\}, \{A = 0.3, B = 0.2\}, \{A = 0.5, B = 0.2\}, \{A = 0.5, B = 0.5\}, \{A = 1, B = 0.2\}, \{A = 1, B = 0.5\}, \{A = 1, B = 1\}\} \).

The algorithm we propose to test \( \alpha \)-WC of a R-CTPP computes a minimal set of meta-scenarios applying Algorithm FST and for each such meta-scenario \( ms \) it checks if the corresponding projection \( Pr(ms) \) is consistent at level \( \alpha \). If the preference functions are semi-convex, in order to test this it is sufficient to test whether the STP obtained from \( Pr(s) \) via its \( \alpha \)-cut (that is considering for each constraint the sub-interval containing elements mapped into a preference \( \geq \alpha \)) is consistent.

If the preference functions are semi-convex the coproblem of \( \alpha \)-WC is \( NP \)-complete since it coincides with deciding if there is an inconsistent STP obtained via the \( \alpha \)-cuts. Thus in such a case testing \( \alpha \)-WC is \( co-NP \)-complete.

**Testing \( \alpha \)-Dynamic Consistency**

In (Tsamardinos, Vidal, & Pollack 2003) the DC of a CTP is checked by transforming the CTP into a Disjoint Temporal Problem (DTP) (Stergiou & Koubarakis 2000) obtained from the union of the STPs corresponding to the projections of the scenarios of the CTP and some additional disjunctive constraints. A CTP is DC if, whenever at certain point in time a given variable must be executed, and it is not possible to distinguish in which scenario we are, there is a value to assign to such a variable which will be consistent with all the possible scenarios that can evolve in future. This means that all the variables representing the same CTP variable in the projections either are constrained to be after observations which allow to distinguish the scenario univocally (and thus can be executed independently of each other) or they must be assigned the same value whenever observation variables do not allow to distinguish the scenarios. This is modeled by adding to the CTP, obtained by the union of all the projections of the CTP, a specific set containing disjunctive constraints (called DC constraints). Briefly, each DC-constraint regarding a variable \( v \) requires that in all scenarios either the execution of \( v \) follows that of all the observation variables of literals in its label, \( L(v) \), or, otherwise, that the occurrences of \( v \) are synchronized.

Adding DC constraints makes the CTP become a DTP (see (Tsamardinos, Vidal, & Pollack 2003) for more details).

Since in R-CTPPs executing a variable at the same time in different scenarios gives the same preference, the reasoning above can be applied directly. In fact, in terms of synchronization only the temporal order matters.

**Theorem 5.** Given an R-CTPP \( Q = (V, E, L, R, OV, O, P) \), let \( D = \langle V', E' \rangle \) be the fuzzy DTP with \( V' = (\bigcup_{Pr(s) = (V,E), s \in MS'} V) \) and \( E' = (\bigcup_{Pr(s) = (V,E), s \in MS'} E) \cup CD \). Then \( Q \) is \( \alpha \)-dynamically consistent if and only if \( D \) is consistent with preference degree \( \alpha \).

Theorem 5 allows us to define an algorithm which, given in input an R-CTPP, tests if it is \( \alpha \)-DC. Such an algorithm first computes the minimal set of meta-scenarios applying Algorithm FST. Next, it tests if the DTP obtained taking the union of the all the STPPs corresponding to projections of meta-scenarios in the minimal set, and adding the \( CD \) constraints, is consistent with optimal preference level \( \alpha \).

Thus the complexity of checking \( \alpha \)-DC is the same as that of solving a fuzzy DTP; we recall that efficient algorithms for finding the optimal preference level of Fuzzy DTPs have been considered in (Feintner & Pollack 2004).

**CTPPs vs. STPPUs**

It is interesting to notice that consistency in CTPs is strongly connected to controllability in STPPs. This arises from the fact that both kinds of problems are concerned with the representation of uncertainty: STPPUs model uncertainty by defining contingent constraints, while CTPs try to capture the fact that both kinds of problems are concerned with the fact that both kinds of problems are concerned with the controllability/consistency of the problem.

We propose here a mapping from STPPUs to CTPP that preserves the controllability/consistency of the problem. The main idea of this mapping is that, if an STPU has contingent constraints defined over finite domains, each possible value that their endpoints can assume is, in a sense, a condition which has been satisfied.

Given an STPU \( Q = (N_r, N_c, L_r, L_c) \), let \( k = |L_c| \), for every soft contingent temporal constraints \( l_i \in L_c \) such that \( l_i = (\alpha_i, b_i) \), \( f_i \) > we discretize the interval \( [\alpha_i, b_i] \) and we denote the number of elements obtained with \( |l_i| \) indicating such a set of elements with \( \{d_{ij}, j = 1 \ldots |l_i|\} \).

For the sake of notation, we write \( I = \{1 \ldots |l_i|\} \) and, for each \( i \in I, j_i = \{1 \ldots |l_i|\} \).
Let us consider the mapping applied to a contingent constraint \( l_i = [a_i, b_i], f_i \), defined on executable \( A \) and contingent variable \( C \). We add \([a_i, b_i]\) observation variables, \( o_{ij} \), and \([l_i]\) variables \( v_{ij} \), one for each possible occurrence of \( C \) at time \( d_{ij} \) in \([a_i, b_i]\). Variable \( o_{ij} \) observes the proposition \( p_{ij} = \{C = d_{ij}\} \), while variable \( v_{ij} \) represents the actual occurrence of \( C \) at time \( d_{ij} \).

Moreover we add a hard temporal constraint with interval \( e_{ij} = [0, 1] \) between \( o_{ij} \) and \( v_{ij} \), and and we add a soft constraint \( e_{oij} = \{d_{ij}, d_{ij}', f_{d_{ij}} > \} \) between \( A \) and \( o_{ij} \).

Any other constraint \( w \) involving \( C \) in the STPPU is replicated \([l_i]\) times, one for each \( d_{ij} \), obtaining constraint \( w_{ij} \) connected to the corresponding \( v_{ij} \) variable.

**Definition 16.** Given an STPPU \( Q = < N_c, N_e, L_r, L_c, > \), where \( I \) and \( J_i \) are as above, we define the CTTP \( C(Q) \) as the tuple \( < V, E, L, R, O, O, P > \), where

- \( P \) is the set of fuzzy atomic propositions \( \{p_{ij}, i \in I, j \in J_i\} \);
- \( V = N_c \cup \{o_{ij}, i \in I, j \in J_i\} \cup \{v_{ij}, i \in I, j \in J_i\}; \)
- \( E = L^r_c \cup \{e_{ij}, i \in I, j \in J_i\} \cup \{e_{oij}, i \in I, j \in J_i\} \cup \{w_{ij}, i \in I, j \in J_i\} \) where \( L_o^c \) is the set of all the requirement constraints in \( L_r \) defined only between executable variables and \( e_{ij}, e_{oij}, \) and \( w_{ij} \) are as defined above;
- \( L : V \rightarrow Q^* \) is a function such that \( L(v_{ij}) = p_{ij} \) and \( true \) otherwise;
- \( R : V \rightarrow FR \) is a function defined as \( R(v_{ij}) = r(0, g) \), where \( g \) is the constant function equal to \( f(d_{ij}) \) where is the preference function of \( l_i \);
- \( O V \subseteq V \) is the set of observation variables \( \{o_{ij} \in I, j \in J_i\}; \)
- \( O : P \rightarrow OV \) is a bijective function such that \( O(p_{ij}) = o_{ij} \).

It is possible to show that this mapping preserves the controllability/consistency notions.

**Theorem 6.** Given an STPPU \( Q \) and its corresponding CTTP \( C(Q) \), \( Q \) is \( \alpha \)-strongly (resp., weakly, dynamically) controllable iff \( C(Q) \) is \( \alpha \)-strongly (resp., weakly, dynamically) consistent.

Notice that the result above mentions \( \alpha \)-weak controllability, which is not defined in (Rossi, Venable, & Yorke-Smith 2006), where only the stronger notion of Optimal-weak controllability is considered. However \( \alpha \)-weak controllability can be directly obtained from the definition of Optimal weak controllability by replacing “optimal” with “\( \geq \alpha \)” whenever the projection has optimal preference at least \( \alpha \).

**References**


Constraint Programming for Planning Routes in an E-learning Environment

Antonio Garrido and Eva Onaindia and Oscar Sapena
Dpto. Sistemas Informaticos y Computacion
Universidad Politecnica Valencia (Spain)
{agarridot, onaindia, osapena}@dsic.upv.es

Abstract
AI planning techniques offer very appealing possibilities for their application to e-learning environments. After all, dealing with course designs, learning routes and tasks keeps a strong resemblance with the main planning components. This paper focuses on planning learning routes under a very expressive constraint programming approach. After presenting a general planning formulation based on constraint programming, we adapt it to an e-learning setting. This requires to model learners profiles, learning concepts, how tasks attain concepts at different competence levels, synchronisation constraints for working-group tasks, capacity resource constraints and multi-criteria optimisation. Finally, we also present a simple example that shows the applicability of this model, the use of heuristics and how the resulting learning routes can be easily generated.

Introduction
Automated planning is an attractive area within AI due to its direct application to real-world problems. Actually, most everyday activities require some type of intuitive planning in terms of determining a set of tasks whose execution allows us to reach some goals under certain constraints. This direct application, the benefits it involves and, finally, the research in planning methods have made it easier the transfer of planning technology to practical applications, ranging from scientific and engineering scopes to social environments. Social environments such as education constitute an attractive field of application because of its continuous innovation and use of ICT. However, it is generally agreed that education has not yet realised the full potential of the employment of this technology. As explained in (Manouselis & Sampson 2002), this is mainly due to the fact that the traditional mode of instruction (one-to-many lecturing, or one-to-one tutoring), which is adopted in conventional education, cannot fully accommodate the different learning and studying styles, strategies and preferences of diverse learners. But now, conventional education is giving way to e-learning environments, which require learners to take the learning initiatives and control how knowledge is presented during instruction (Atolagbe 2002). Particularly, many European countries signed the Bologna joint declaration of the European space for higher education1, which entails an important change in the learning process. With this declaration, learners roles are much more dynamic, active and autonomous.


The amount of one-to-many lecturing decreases and significantly increases the amount of self-learning through the construction of coherent learning routes according to a certain instructional course design. Finally, this course design recommends sequence of educational tasks and material, tailored to individual learners needs and profiles.

In this paper we address the construction of learning routes from the viewpoint of planning based on constraint programming. After all, generating a learning route represents a planning activity with the following elements: learning goals to be attained, profile-adapted tasks with their prerequisites and learning outcomes (i.e. preconditions and effects, respectively), non-fixed durations, resources, ordering and synchronisation constraints, and collaboration/cooperation relations. The underlying idea is to plan a learning route for a learner with a given profile in order to reach some learning goals. Each route consists of a sequence of tasks, such as attending an in-person lesson, doing a lab exercise, writing a report, etc. Although intuitively each course-plan is initially created individually for each given learner, there are some particular tasks that need to be done simultaneously by several learners, such as attending a lab for the same practice work. Additionally, these tasks may require some type of synchronisation (for instance, doing a working-group task), where the start and/or end must happen at the same time. In this context, the planning component is not particularly costly since the plan is usually small (the number of tasks is between 10-20 per route, though there may be a lot of alternatives). On the contrary, the scheduling component is more significant because of the resource availability, the diversity of constraints and their handling and synchronisation among different routes. These features are not easily included in traditional planning as they require artificial mechanisms to be managed, which complicate the planning algorithms. For instance, a very frequent type of constraints in an e-learning scenario such as synchronisation...
constraints, where several actions need to meet throughout a whole interval, is not easily represented and handled in planners.

Our approach for planning learning routes relies on the constraint programming formulation presented in (Garrido, Onaindia, & Arangu 2006), based on (Vidal & Geffner 2006), which encodes all type of constraints derived from both planning and scheduling features. Such a formulation provides a high level of expressiveness to deal with all the elements required in an e-learning setting and has several advantages: i) it is a purely declarative representation and, consequently, can be solved by any type of CSP solver; ii) although the formulation is automatically derived from the course design, specific ad-hoc control information in the form of hand-coded domain knowledge or domain-dependent heuristics can be included in the formulation to make the resolution process more efficient; and iii) optimality is a major issue in this context and so different optimization criteria can be defined w.r.t. the number of actions of the learning routes, the duration of their tasks or the cost associated to them. In summary, this paper introduces a formulation of planning problems by means of constraint programming and the application of such a formulation to solve a learning-route planning problem. This work is being developed under an on-going national research project whose objective is the application of AI planning techniques for the automated design, execution, monitoring and evaluation of learning routes.

This paper is organised as follows. In the second section we present the e-learning environment and its relation to AI planning, motivating some needs for using a constraint programming approach. The third section briefly reviews the formulation of a planning problem by means of constraint programming, while in the fourth section this formulation is adapted to fit an e-learning scenario. In the fifth section an example of application is analysed, showing part of the formulation, implementation and results. Finally, we present the conclusions of the paper.

E-learning and AI Planning

The application of AI planning techniques has reported important advances in the generation of automated courses within e-learning. One of the first attempts in this direction was the work in (Peachy & McCalla 1986), in which the learning material is structured in learning concepts and prerequisite knowledge is defined, which states the causal relationship between different concepts. This instructional method was one of the first approaches to combine instructional knowledge and artificial intelligence planning techniques to generate sequences of learning materials. In the same direction, Vassileva designed a system that dynamically generates instructional courses based on an explicit representation of the structure of the concepts/topics in the domain and a library of teaching materials (Vassileva 1997). Other approaches have introduced hierarchical planners to represent pedagogical objectives and tasks in order to obtain a course structure (Ullrich 2005). Most recent works, such as the one presented in (Vrakas et al. 2007), incorporate machine learning techniques to assist content providers in constructing learning objects that comply with the ontology concerning both learning objectives and prerequisites.

Although the task of designing learning routes for e-learning environments has been accomplished from different perspectives, one of the most intuitive approaches is the instructional planning design (van Marcke 1992), which is based on learning outcomes, information processing analysis and prerequisite analysis (Smith 2005). Therefore, e-learning can be considered as a particular planning domain with specific constraints.

The underlying idea about instructional planning is to provide a constructivist learning strategy based on both instructional tasks and instructional methods, i.e. representations of different routes to achieve the goals. An expert user, usually a teacher or instructor, designs a course as a set of learning tasks with prerequisites that are required prior to the execution of the task, learning outcomes that are attained after the execution and a positive duration. The task duration is usually a non-fixed value because in a learning environment this value cannot be precisely determined in advance as it varies among learners. Additionally, the instructor can define task-task and/or task-outcome constraints as well as deadlines to attain the learning goals. The main aim of an instructional session is to find a valid learning route for a learner to achieve the learning goals, on the basis of his/her particular constraints (e.g. personal profile, previous knowledge, resource availability and temporal constraints). In other words, an instructional design determines which learning tasks are present and in which order by using a structure of concepts/topics and tasks. As can be noted, an instructional design keeps a strong resemblance with a plan, as it is usually modelled in AI planning. Analogously, the elements necessary to find this design are similar to the elements defined traditionally in planning, since tasks can be expressed in terms of actions with duration, prerequisites and effects.

First, the course design defined by the instructor may be seen as a planning domain which contains the tasks that can be used to form the final learning routes. There is, however, slight differences in the way the e-learning domain knowledge is represented. While a planning domain is represented as a plain–text file (e.g. PDDL format (Gerevini & Long 2006)), a course design is commonly represented in a graphic way, thus explicitly modelling the structural relations between tasks. The reason for this is clear: the instructor who defines the course design neither needs being an expert in route modeling languages nor being interested in their syntax details. On the contrary, a graphic representation turns out more useful when showing the workflow among tasks (see Figure 1 below).

Second, a learning task is equivalent to an action. A task has prerequisites and learning outcomes, analogously to the conditions and effects of an action. Although a non-fixed duration is not a common feature in planning, where the du-
ration of the actions is well-known, this does not involve any difficulty in a constraint programming setting as it can be simply modelled by creating a new variable in the problem that represents such a duration (see next section for more details). Regarding the definition of constraints, there exist two basic types: task-task and task-outcome ordering constraints and/or deadlines. These types of constraints are not commonly used in planning (particularly, orderings are only due to causal link relations) but their definition within constraint programming is straightforward. As we will discuss later, other more elaborate constraints can be defined similarly and easily, which makes a constraint programming setting more appealing than traditional planning.

Third, prerequisites and learning outcomes, also known as knowledge objects or simply as concepts, can be seen as fluents that are required/achieved by tasks. In planning, fluents can represent propositional or numeric information. In the former, the domain is binary: the fluent (as a boolean proposition) is either present or not. In the latter, the domain can be defined as a real or integer domain, where the range of possible values is significantly higher than two. In an e-learning environment all concepts are numeric because no concept is entirely boolean; when a learner performs a task and achieves a concept it is not as easy as getting or not getting such a concept, but several degrees of achievement can be considered. This means that all the planning process has to reason with numeric information in order to attain concepts at a certain competence level, such as achieving a given concept in a degree greater than 5. The way to achieve these concepts involves a subtle particularity as well. In traditional planning, actions modify their numeric effects in several ways by assigning a new value, increasing, decreasing or scaling its current value. However, in an e-learning domain tasks only have increasing effects, i.e. tasks can only improve the value of a concept but never worsen it. Using exclusively increasing effects means that once a learner has attained a concept through the execution of one or several tasks, no further tasks can lessen the competence level the learner has attained; that is, the learning process is a monotonically incremental process. Some authors\textsuperscript{3} agree that performing tasks may alter the current knowledge state (value of the concepts) of the learners and even defeat some concepts already attained in the past. In a constraint programming formulation this is possible by simply expressing such an alteration as a threat to be solved. However, we consider the fact of forgetting concepts, i.e. expressing limited persistence on the concepts, more appealing for a real e-learning environment. This way, we may include a concept-task temporal constraint in the form: a concept can only be used as a prerequisite for a task within 40 hours of its achievement. This type of constraint can be easily modelled using the constraint programming formulation defined below.

Finally, quality assessment for learning routes plays a similar role to the optimisation process in planning. Now, the optimisation criterion can be seen from two different perspectives. From the learner’s point of view, the criterion to be optimised is the length of the learning route (plan), either in terms of number of tasks, their duration or difficulty level. It is important to note that the learning route of a learner is formed by a set of sequential tasks. In planning terminology this implies having a sequential plan per learner, where no tasks are executed simultaneously. Obviously, a learner cannot perform two tasks at the same time, but it is possible and frequent to have parallel plans for different learners. From the expert or teaching centre point of view, the criterion to be optimised may be associated to the cost of the tasks, usually given in terms of the cost of the used resources. For instance, if some tasks need an expensive resource the optimisation criterion will tend to reduce the usage cost by using alternative cheaper tasks. The optimisation of resource usage and cost is more a scheduling feature than a planning one itself, but a constraint programming formulation can combine both features under the same model, which again makes a constraint programming setting an interesting approach. Additionally, a multi-criteria optimisation function that combines the two viewpoints can be easily defined to take into consideration a more representative metric.

**Planning as a Constraint Programming Formulation**

In this section we present a general model to formulate planning as constraint programming that will be used as a basis for the e-learning scenario. Constraint programming formulations have been used in many approaches to handle both planning and scheduling features. A common feature that appears in these approaches is that they rely on constraint satisfaction techniques to represent and manage all different types of constraints, including the necessary constraints to support preconditions, mutex relations and subgoal preserving. That is, they use constraint programming for planning by encoding a planning problem as a CSP. Therefore, CSP formulations for planning include reasoning mechanisms to represent and manage the causal structure of a plan as well as constraints that denote metric, temporal and resource constraints. This general formulation through constraint programming has the ability to solve a planning problem with very elaborate models of actions. Moreover, automated formulations, like the ones presented in (Vidal & Geffner 2006; Garrido, Onaindia, & Arangu 2006) have the advantage that once the constraint programming model is formulated, they can be solved by any CSP solver.

In a constraint programming setting, a problem is represented as a set of variables, a domain of values for each variable and a set of constraints among the variables. Variables are basically used to define actions and conditions, both propositional and numeric, required by actions, along with the actions that support these conditions and the time when these conditions occur (time is modelled in \( \mathbb{R} \)). Variables are defined for each action present in the problem, which may comprise all actions of the planning domain (after grounding all operators), or a smaller subset. Every action \( a \) is represented by the following basic variables (Garrido, Onaindia, & Arangu 2006):

- \( S(a) \), \( E(a) \) represent the start and end time of action \( a \).

\textsuperscript{3}Dimitris Vrakas (2007), personal communication.
• \(dur(a)\) represents the duration of action \(a\).

• \(InPlan(a)\) encodes a binary variable that denotes the presence of \(a\) in the solution plan.

• \(Sup(c, a)\) represents the action that supports condition \(c\) for action \(a\), where \(c\) represents a fluent that can be either propositional or numeric.

• \(Time(c, a)\) represents the time when the causal link \(Sup(c, a)\) happens; if \(c\) is a numeric condition, \(Time(c, a)\) represents the time when the action in \(Sup(c, a)\) last updates \(c\).

• \(Req\_{start}(c, a)\) and \(Req\_{end}(c, a)\) represent the interval in which action \(c\) requires condition \(c\). These variables provide a high expressiveness for representing a wide type of conditions, from punctual conditions to conditions required throughout an interval beyond the action duration.

• \(V_{\text{actual}}(c, a)\) is a variable only used for numeric conditions which denotes the value of the corresponding fluent \(c\) at time \(Req\_{start}(c, a)\) if \(a\) requires condition \(c\); otherwise, this variable stores the actual value of the numeric fluent at time \(S(a)\).

• \(V_{\text{updated}}(c, a)\) represents the new value for the fluent \(c\) updated by action \(a\). This variable is only necessary in case \(a\) modifies the fluent \(c\).

Constraints represent relations among variables and correspond to assignments and bindings of the variables, supporting, temporal and numeric constraints. The basic constraints defined for each variable that involves action \(a\) are:

• \(S(a) + dur(a) = E(a)\) binds the variables start and end of any action \(a\).

• \(E(\text{Start}) \leq S(a)\) represents that any action \(a\) must start after fictitious action \(\text{Start}\).

• \(E(a) \leq S(\text{End})\) represents that any action \(a\) must finish before fictitious action \(\text{End}\).

• \(Time(c, a) \leq Req\_{start}(c, a)\) forces to satisfy condition \(c\) (either propositional or numeric) before it is required.

• \(Time(c, a)\) represents the time when the action in \(Sup(c, a)\) adds or updates (propositional or numeric) condition \(c\). When \(c\) is a numeric condition, \(V_{\text{actual}}(c, a) = V_{\text{updated}}(c, Sup(c, a))\).

• \(Cond(c, a) = V_{\text{actual}}(c, a) \text{ comp-op expression}\), where \(\text{comp-op} \in \{<, \leq, =, \geq, >, \neq\}\) and expression is any combination of variables and/or values that is evaluated in \(\mathbb{R}\), which represents the condition that the corresponding fluent \(c\) must satisfy in \([Req\_{start}(c, a), Req\_{end}(c, a)]\) for action \(a\).

• Branching. \(Sup(c, a) = b_i \land Sup(c, a) \neq b_j \mid \forall b_i, b_j (b_i \neq b_j)\) that supports \(c\) for \(a\), which represents all the possibilities to support \(c\), one for each \(b_i\) (while \(|Sup(c, a)| > 1\).

• Solving threats. Let \(time\_threat(b_i)\) be the time when action \(b_i\) threatens the causal link \(Sup(c, a)\), i.e., when \(b_i\) changes the value of \(c\) generated by \(Sup(c, a)\). In that case, the constraint \((time\_threat(b_i) < Time(c, a)) \lor (Req\_{end}(c, a) < time\_threat(b_i))\) must hold, which represents the idea of threat resolution by promotion or demotion.

• Solving mutexes. Let \(time(b_i, c)\) and \(time(b_j, c)\) (\(b_i \neq b_j\)) be the time when \(b_i\) and \(b_j\) modify \(c\), respectively; if \(c\) is a propositional fluent \(b_i/b_j\) generates/deletes \(c\), whereas if \(c\) is a numeric fluent \(b_i\) and \(b_j\) give different values to \(c\). Hence, \(\forall b_i, b_j: time(b_i, c) \neq time(b_j, c)\) must hold, which represents the mutex resolution between the two actions being executed in parallel: \(b_i\) and \(b_j\) cannot modify \(c\) at the same time.

This flexible formulation also admits the specification of complex planning constraints such as persistence of concepts, temporal windows in the form of external constraints or general customised n-ary constraints to encode complex constraints that involve several variables of the model. Note that despite the high number of constraints in the model, they are only essential when involving actions with their variables \(InPlan() = 1\).

The expressiveness of this model formulation facilitates the encoding of any planning problem, from purely propositional problems to more elaborate domains which mix propositional and numeric information, along with more complex constraints. In general, the resolution of the constraint programming model is a hard task, which becomes even more difficult when there exists many variables to be instantiated and constraints to fulfill. However, the most costly task in the overall resolution process is selecting the values for variables \(Sup(c, a)\), i.e., establishing the causal links of the actions that create the causal structure of the plan. On the contrary, this formulation model shows very efficient when the main aim is only to schedule plans or solve problems with medium/low load of planning. In this case, variables \(InPlan()\) are already instantiated; the plan is already known and the only task is to assign the execution times of the actions and satisfy all problem constraints. In other words, this model turns out to be very appropriate in those problems where planning is not the big deal, that is, for pseudo-planning problems with a high load of scheduling.

Regarding the formal properties of this formulation, it inherits the properties of a POCL approach such as soundness, completeness and optimality. Soundness and completeness are guaranteed by i) the definition of the model itself, because all the alternatives to support causal links and solve threats and mutexes are considered, and ii) the completeness of the CSP solver, which performs a complete exploration of the domain of each variable. Optimality is also guaranteed by the CSP solver by performing an exhaustive, complete search until finding the best quality solution.

### Adapting the Constraint Programming Formulation for E-learning

The general constraint programming formulation presented in the previous section needs to be slightly changed in order to be adapted to an e-learning environment. On the one hand, it needs to be simplified for all features related to: i) variables and constraints for propositional information, which are not used in this environment, ii) variables
Elimination of $\text{Req}_{\text{start}}$, $\text{Req}_{\text{end}}$. The constraint $Time(c,a) \leq \text{Req}_{\text{start}}(c,a)$ now becomes $Time(c,a) \leq S(a)$.

Sequential plan per learner: let $T_i$ be the set of all possible tasks that a learner $l_i$ could execute. The constraint $\forall t_j, t_k (t_j \neq t_k) \in T_i : (E(t_j) \leq S(t_k)) \lor (E(t_k) \leq S(t_j))$ must hold.

Synchronisation of working-group tasks: let \{t_{1,1}, t_{1,2}, \ldots, t_{1,n}\} be the tasks that learners $l_{1,1}, l_{1,2}, \ldots, l_{1,n}$ must respectively execute at the same time as a common working-group task. The constraint $(S(t_{1,1}) = S(t_{1,2}) = \ldots = S(t_{1,n})) \land (E(t_{1,1}) = E(t_{1,2}) = \ldots = E(t_{1,n}))$ must hold (obviously all the durations must be the same). Additionally, if these tasks require a particular resource $R_j$, such as a lab, special equipment, etc., they need to fit in the temporal window of the resource availability given by $[\min(tw(R_j)), \max(tw(R_j))]$. This way, the next constraint must also hold: $(\min(tw(R_j)) \leq S(t_{1,1})) \land (E(t_{1,1}) \leq \max(tw(R_j)))$.

Resource capacity: let $T = \{t_{i,1}, t_{i,2}, \ldots, t_{i,n}\}$ be the set of all tasks that are executed simultaneously (all the starting and ending points coincide, respectively) by different learners and require a resource $R_j$. Assuming that these tasks consume some quantity of a resource $R_j$ (denoted by $use(t_{i,1}, R_j)$), the next constraint to avoid resource overconsumption must hold: $\sum_{i=1,n} use(t_{i,1}, R_j) \leq C(R_j)$, where $C(R_j)$ is the max capacity of resource $R_j$. This ensures that at any time throughout the execution of the tasks in $T$, the sum of all the individual resource consumption of the tasks does not exceed the resource capacity.

Domain-dependent heuristics

When solving a planning problem, the use of adequate heuristics becomes essential to improve the efficiency of the search and, consequently, the overall performance. Clearly, the same happens when solving a constraint satisfaction problem and especially when the problem represents a planning problem that contains many variables and constraints. One of the most effective points to apply heuristics is the branching point, i.e. when the CSP solver needs to assign a value to the variables $InPlan()$, $Sup()$ and $S()$/$E()$ (note that the value of the remaining variables of the model comes from a propagation of the assignment for these variables). In this point, the heuristics can be defined as the usual variable and value selection heuristics in order to reduce the branching factor, that is, which variable to select first and which value to instantiate first, respectively. Traditionally, CSP heuristics use domain-independent information to estimate this selection order, such as first select the variable with the max number of constraints, or the one with the min domain, or instantiate the values in an increasing, decreasing or random order. However, this may not be the best approach to tackle an e-learning planning problem as can be seen in the next example.

Let us assume an e-learning setting with the tasks $T_1 = \{t_{1,1}, t_{2,1}, \ldots, t_{1,n}\}$ and $T_2 = \{t_{1,2}, t_{2,2}, \ldots, t_{2,k}\}$ that can be included in the learning routes (plans) for learners $l_1$ and $l_2$, respectively. Since the aim of the CSP solver is to find which tasks will be part of the solution, when using a blind heuristic the variable selection could try to instantiate first the variables associated with task $t_{1,1}$, then $t_{1,2}$, $t_{2,1}$, $t_{2,2}$ and so on, i.e. alternating tasks of the two different learners in a breadth-first strategy. If there appears a conflict because of the selected tasks for a learner, a lot of unnecessary backtracking will be done on the tasks of the other learner. For instance, if task $t_{1,1}$ was wrongly chosen, the CSP solver will need to backtrack on the already-instantiated tasks $t_{1,2}$, $t_{2,2}$, etc. that will not fix the problem of learner $l_1$ and will imply a lot of thrashing. This indication of inefficiency is much more significant when the number of learners increases and, particularly, in problems with symmetry, where a lot of effort is wasted trying to unsuccessfully instantiate almost identical tasks for different learners. Although there are some works about more efficient ways to guide backtracking, learn from conflicts and exploit symmetry when planning as a CSP (Kambhampati 2000; Zimmerman & Kambhampati 1999), we can apply a very effective domain-dependent heuristic by simply grouping the variables w.r.t. the learner they belong to. Thus, the selection strategy selects first the variables of one learner and does not move to a second learner until finding a valid learning route for the first one. This can be seen as a depth-first strategy, which though it does not avoid backtracking in conflicting cases with different learners sharing the same oversubscribed resource, it shows very effective in most situations. Actually, this simple strategy allows to find more learning routes for more learners in less time. Further, this heuristic has two additional advantages: i) it is valid for any CSP solver, and ii) it does not introduce an overhead in the solving process as the variable grouping can be computed before solving the problem, i.e. the grouping is independent of the solving process itself.

An E-learning Scenario of Application

In this section we present an e-learning scenario that will be used as an application example to show how to plan learning routes under a constraint programming approach. We assume that an expert defines the course design depicted in Figure 1, which consists of 7 concepts (6 + 1 previous concept) and 9 tasks of different non-fixed duration. Each concept represents a learning object, i.e. a knowledge item that can be attained from one or more tasks. Each task may rep-
Figure 1: Course design for the e-learning scenario of application. Some tasks and concepts generated as effects are profile-dependent. The duration of the tasks is indicated between brackets. Note that tasks and concepts represent the idea of actions, preconditions and effects used in planning.
resent a discrete lesson, a seminar activity, a public talk or even a higher level course. For simplicity matters we assume that each learner can execute each task only once. The course shown in Figure 1 is appropriate for learners with different learning styles, such as input profile (visual or verbal) or organisation profile (inductive or deductive), following the classification given in (Felder & Silverman 1988). According to the learner’s profile, (s)he can perform, or not, a given task. Particularly, Task2 is only adequate for learners with a visual input profile. Moreover, tasks attain concepts at different competence levels (percentages) depending on the type of profile they are applied to. For instance, Task3 generates Concept4 at different levels if the learner is visual or verbal. We also assume that Task1 is an in-person lesson that needs to be performed in a synchronised way for all the learners, while Task4 requires a particular resource of max capacity 2, i.e. only two learners can perform such a task at the same time.

We apply the previous design course in an e-learning scenario with 4 learners with different profiles. Table 1 shows the profiles for these learners, the initial values of prev-Concept1 and the value required for Concept6, which is considered as the final learning goal to be attained at different competence levels.

<table>
<thead>
<tr>
<th>Learner</th>
<th>Profile</th>
<th>prev-Concept1</th>
<th>Concept6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learner1</td>
<td>visual, inductive</td>
<td>= 25</td>
<td>≥ 100</td>
</tr>
<tr>
<td>Learner2</td>
<td>verbal, inductive</td>
<td>= 0</td>
<td>≥ 75</td>
</tr>
<tr>
<td>Learner3</td>
<td>verbal, deductive</td>
<td>= 50</td>
<td>≥ 100</td>
</tr>
<tr>
<td>Learner4</td>
<td>visual, deductive</td>
<td>= 0</td>
<td>≥ 50</td>
</tr>
</tbody>
</table>

Table 1: Initial features of the four learners.

Formulation of the problem According to the constraint programming model presented above, the formulation for Task2 (and its concepts) of Learner1 includes the following variables and constraints:

- \( \text{InPlan}(T2) \in [0, 1] \)
- \( S(T2), E(T2) \in [0, \infty[ \)
- \( \text{dur}(T2) \in [3, 5] \)
- \( \text{Sup}(C1, T2) \in \{\text{Start}, T1\} \)
- \( \text{Sup}(C3, T2) \in \{\text{Start}, T3, T4, T5\} \)
- \( \text{Sup}(C4, T2) \in \{\text{Start}, T3, T4\} \)
- \( \text{Time}(C1, T2), \text{Time}(C3, T2), \text{Time}(C4, T2) \in [0, \infty[ \)
- \( S(T2) + \text{dur}(T2) = E(T2) \)
- \( E(\text{Start}) \leq S(T2) < E(T2) \leq S(\text{End}) \)
- \( \text{Time}(C1, T2) \leq S(T2), \text{Time}(C3, T2) \leq S(T2), \text{Time}(C4, T2) \leq S(T2) \)
- if \( \text{Sup}(C1, T2) = \text{Start} \) then \( \text{Time}(C1, T2) = \text{Start} \)
  if \( \text{Sup}(C1, T2) = T1 \) then \( \text{Time}(C1, T2) = T1 \)
  if \( \text{Sup}(C3, T2) = \text{Start} \) then \( \text{Time}(C3, T2) = E(T3) \)
  if \( \text{Sup}(C3, T2) = T4 \) then \( \text{Time}(C3, T2) = E(T4) \)
  if \( \text{Sup}(C3, T2) = T5 \) then \( \text{Time}(C3, T2) = E(T5) \)
- if \( \text{Sup}(C4, T2) = \text{Start} \) then \( \text{Time}(C4, T2) = \text{Start} \)
  if \( \text{Sup}(C4, T2) = T3 \) then \( \text{Time}(C4, T2) = E(T3) \)
  if \( \text{Sup}(C4, T2) = T4 \) then \( \text{Time}(C4, T2) = E(T4) \)
- \( \forall T_i \ (T_i \neq T2) \) of Learner1: \( (E(T2) \leq S(T_i)) \lor (E(T_i) \leq S(T2)) \)

The variables for other tasks of Learner1 and other learners are generated similarly, including the synchronisation and resource capacity constraints if necessary.

Implementation and results The constraint programming formulation is modelled and solved by Choco. We have modified the variable selector of the Choco engine to take into consideration the problem variables grouped by learner, analysing first Learner1, then Learner2 and so on. The metric to optimise involves an expression with as many variables as the user requires, such as number of tasks, cost for using the labs or any combination of them. In this example we perform the optimisation task from the learner’s point of view, i.e. optimising the number of actions in the four learning routes. This means to find the solution with the min number of tasks to reach the learning goals, which in this case also coincides with the routes of the shortest makespan. In such a case, Choco performs an exhaustive search of solutions until no feasible solution improves the quality of the best solution found until that time. In a problem like this, guaranteeing the optimal solution is a very expensive task, but in most cases a good solution can be found in a few seconds. Actually, Choco finds a solution very quickly and this turns to be the optimal one, though this cannot be generalised. Particularly, in this problem Choco found two solutions, the first one with 21 tasks and the second with 20, in 1 and 12 seconds respectively. Although we extended the search for more than 20 minutes no better solution was found. It is important to note that the used heuristic plays a valuable role in the solving process. For instance, we tried to solve this problem using the Choco default variable selection heuristic (first select the variable with the min domain), and the first solution was found after 12 minutes (the optimal solution took nearly 15 minutes). This shows that very simple heuristics can improve a lot the performance, with no changes in the constraint formulation at all.

The learning routes for each learner are shown in Figure 2. The longest route has a makespan of 16 (Learner1), but other learners’ routes are shorter. As indicated in the problem constraints, it is important to note two properties: i) Task1 is executed at the same time by the four learners because of the synchronisation requirement of an in-person lesson, and ii) Task4 can be executed in parallel at most by two learners because of the capacity constraint.

\(^3\)Choco is a Java library for constraint satisfaction problems that can be downloaded from http://choco.sourceforge.net
 Constraint programming formulation is a very appropriate approach to tackle planning problems that require the representation and management of a wide range of constraints, as it is the case of e-learning environments. Designing a learning route can be viewed as generating a plan in a domain where it is necessary to handle resources and the capability of such resources, synchronised tasks among several learners, orderings between tasks, deadlines or other customised and elaborate constraints. In principle, as a planning problem, the design of learning routes could be accomplished by using a current state-of-the-art planner. However, current planners cannot afford complex constraints as task synchronisation or, otherwise, it would be necessary many artificial tricks for representing and handling it. On the other hand, optimality is a major issue in e-learning contexts either from the learner or teaching center viewpoint, so it is important to guarantee good-quality plans as this might affect the overall learning route.

In an e-learning context, the activity that requires a little bit of effort is the definition of the course design by the instructor. It is necessary to classify the learning tasks, study the prerequisites and outcomes of each task (concepts), identify the profiles for which the task is best focus, determine the assessment points, establish the competence level for the concepts, etc. However, although this can be a bit tedious, the course design comprises the specification of a learning domain that can be used for the generation of learning routes in many different contexts (teaching centers), for different learner profiles and with different optimisation criteria. In other words, it is worth the effort devoted to course design in favor of the reusability degree we can obtain with our approach for planning routes in e-learning environments.

We can conclude by saying that the expressiveness of constraint programming makes it very appropriate for the modelling learning routes. The adequacy is also given by the fact that designing learning routes is not a very complex planning problem but rather a complex scheduling problem. For this reason, constraint programming approaches seem to be a promising direction for e-learning contexts.

Conclusions

Acknowledgements

This work has been partially supported by the Spanish government project MCYT TIN2005-08945-C06-06 (FEDER), by the Valencian government project GV06/096 and by the Universidad Politecnica de Valencia under the PAID-04-07 programme.

References


Extensions of the COMPLETION Constraint

András Kovács
Projet Contraintes, INRIA Rocquencourt, France and
Computer and Automation Research Institute, Hungary
akovacs@sztaki.hu

J. Christopher Beck
Department of Mechanical & Industrial Engineering
University of Toronto
jcb@mie.utoronto.ca

Abstract
The COMPLETION global constraint has been proposed for single-machine, unary-resource, total weighted completion time scheduling problems where it has shown good performance. In this paper, we look at extending the constraint in two ways. First, we apply the constraint to multiple machine scheduling problems, in the form of job shop scheduling. It is shown that under the right allocation of weights to activities, the COMPLETION constraint results in significantly better scheduling performance compared to the standard expression of the weighted completion time. Second, we extend the constraint from the unary to discrete resources. Empirically this extension results in an orders of magnitude improvement in the number of nodes required to find a solution though a somewhat more mixed result on run-time.

Introduction
The COMPLETION constraint is a recently proposed global constraint to propagate the total weighted completion time of activities on a single unary capacity resource (Kovács & Beck 2007). It employs a lower bound based on a preemptive relaxation that is computable in polynomial time and a recomputation of the lower bound to prune values from the start time domains of the activities. Empirical results have shown strong performance compared to existing CP approaches on single machine problems with release times. On the other hand, state-of-the-art techniques developed for the specific single machine problem, including a branch-and-bound search with powerful dominance rules and a sophisticated dynamic programming approach, are still significantly better than the COMPLETION constraint with basic CP tree search. The strength of the COMPLETION constraint is its applicability to problems with various side constraints, which are not tractable by the above dedicated methods.

In this paper, we investigate two extensions of the COMPLETION constraint:

1. Applying COMPLETION to multiple machine scheduling;
2. Extending COMPLETION to discrete resource scheduling.

Previous Work
A critical component of the success of pure constraint programming (CP) techniques to optimization problems is the ability to design a model that exhibits significant back propagation. Back propagation is the reduction in search space through pruning of the domains of decision variables as a result of a new bound on the optimization function. Models that exhibit a high degree of back propagation will tend to be successful as new (sub-optimal) solutions will result in a smaller subsequent search space. In contrast, without back propagation, the full search space will need to be explored, suggesting that CP will not result in any better performance than any other search technique.

The significance of cost-based global constraints for strong back propagation has been emphasized by Focacci et al. (Focacci, Lodi, & Milano 2002b). Cost-based constraints with effective propagation algorithms include the global cardinality constraint with costs (Régis 1999), the minimum weight all different constraint (Sellmann 2002), the path cost constraint for traveling salesman problems (Focacci, Lodi, & Milano 2002a), and the cost-regular constraint (Demassey, Pesant, & Rousseau 2006). The COMPLETION constraint is a cost constraint for total weighted completion time. This objective and other “sum-type” objectives are common in scheduling applications.

The single-machine, unary capacity total weighted completion time problem when activities have different release dates is NP-hard (Chen, Potts, & Woeginger 1998). Using the classical $\alpha|\beta|\gamma$ scheduling notation (Graham et al. 1979) with $w_i$ being the weight of job $A_i$, $r_i$ its release time, and $C_i$ being its completion time in the schedule, this problem can be expressed as $1|r_i|\sum w_iC_i$. However, a preemptive version of the problem with a slightly modified objective function is computable in polynomial time and serves as a lower bound. Let $M_i$ be the mean busy time of activity $A_i$. That is, $M_i$ is the mean point in time at which the machine is busy processing activity $A_i$. The problem $1|r_i,pmtn|\sum w_iM_i$ can be solved in $O(n \log n)$ where $n$ is the number of activities (Goemans et al. 2002).
We can, therefore, implement a global constraint that filters the domains of the start time variables by computing the cost of the optimal preemptive mean-busy time relaxation for each activity \( A_i \) and each possible start time \( t \) of activity \( A_i \), with the added constraint that activity \( A_i \) must start at time \( t \). If the cost of the relaxation is greater than the current best known solution, then \( t \) is removed from the domain of the start time variable of \( A_i \).

In practice, we do not naively re-solve the relaxation for each start time in the domain of each activity. Instead, a much faster algorithm allows us to transform an initial relaxed solution into preemptive schedules with given start time assignments. For a detailed presentation of this algorithm and the COMPLETION constraint, in general, readers are referred to (Kovács & Beck 2007).

Formally, the COMPLETION constraint is defined as follows:

\[
\text{COMPLETION}(\{S_1, \ldots, S_n\}, [p_1, \ldots, p_n], [w_1, \ldots, w_n], C)
\]

where there are \( n \) activities, \( A_i \), to be executed without preemption on a single, unary resource. Each activity is characterized by its processing time, \( p_i \), and a non-negative weight, \( w_i \). The start time variable of \( A_i \) is denoted by \( S_i \). The total weighted completion time of the activities will be denoted by \( C \). We assume that all data are integral. The constraint enforces \( C = \sum_i w_i(S_i + p_i) \).

**Application to Multiple Machine Scheduling**

An \( n \times m \) job shop scheduling problem (JSP) has \( n \) jobs each composed of \( m \) completely ordered activities. Each activity requires exclusive use of one resource during its execution. The duration and the resource for each activity are given and may be different from that of other activities: often, as in the problems studied here, a different resource is specified for each activity in a job. An activity cannot start until the activity immediately preceding it in the same job has finished. The standard JSP decision problem asks if, for a given makespan, \( D \), all activities can finish by \( D \). This is a well-known NP-complete problem (Garey & Johnson 1979). It is not uncommon to solve the optimization version of the JSP with the goal of minimizing makespan or some other metric such as sum of earliness and tardiness (Beck & Refalo 2003) or, in the present case, the sum of the weighted completion time of all jobs. More formally, given a set of jobs \( J \) and a weight, \( w_j, j \in J \), our goal is to find a start time for each activity such that:

- no resource executes more than one activity at a time
- each activity starts after its preceding activity in the job-order ends
- \( \sum_{j \in J} w_jC_{E_j} \) is minimized, where \( E_j \) is the last activity in job \( j \).

We study square JSPs (i.e., \( n = m \)) where each job has exactly one activity on each resource.

**From Job Weights to Activity Weights**

Applying the COMPLETION constraint to the JSP is straightforward as the resources have unary capacity. Therefore, we can apply the constraint to each resource, individually. The only complication is that the constraint uses a weight on each activity and the JSP has weights on each job. Therefore, we need to define a mapping from job weight to activity weight.

The obvious approach, which we refer to as last, is to assign the job weight to the last activity in each job and to assign all other activities a weight of zero. We then place a COMPLETION constraint on each resource that has a non-zero weight activity and the total weighted completion time is the sum of the \( C \) values of each COMPLETION constraint.

The last approach has two main drawbacks. First, a computationally expensive COMPLETION constraint is placed on each resource. Second, the COMPLETION constraint makes inferences based on a relaxation that focuses on the interaction among activities on the same resource. Clearly, this interaction is not captured when the weighted activities are on different resources. In the extreme, the last activity in each job may be the only weighted activity on a resource. Under such circumstances, the COMPLETION constraint is not able to make any inferences stronger than the simple weighted sum constraint.

Therefore, we propose a different weight mapping, called busy. Before solving, we identify the most loaded resource, i.e., the “busy” resource, by summing the durations of the activities on each resource and selecting the resource with highest sum. The weight of each job is assigned to the last activity of the job that is processed on the busy resource. All other activities have a weight of zero. A single COMPLETION constraint can then be posted on the busy resource. To calculate the total weighted completion time, we need to correct for the fact that the weighted activity is not necessarily the last activity in the job.

Formally, as above, let \( E_j \) be the last activity in job \( j \) and let \( B_j \) be the single weighted activity in job \( j \). Our optimization function is then: \( C + \sum_{j \in J} w_j(C_{E_j} - C_{B_j}) \) where \( C \) is the cost variable associated with the COMPLETION constraint.

**Experimental Details**

To test the effectiveness of the COMPLETION constraint, we compare it against the standard weighted sum, \( WS \), form of the optimization function. For completeness, we also run \( WS \) with last and busy weight allocations.

We experiment with two styles of search: chronological backtracking and randomized restart. For chronological backtracking (i.e., depth-first search) we use a customized version of the SetTimes heuristic available in ILOG Scheduler 6.3. The heuristic selects all activities with minimum start time and breaks ties by choosing the activity with the highest ratio of weight to duration. The selected activity is scheduled at its earliest start time. Upon backtracking, the scheduled activity is postponed, meaning that it will not be considered for selection again until constraint propagation.
has increased its minimum start time. When all minimum start time activities are postponed, the search backtracks further as no better schedule exists in the subtree.

For randomized restart, the limit on the number of backtracks before restarting evolves according to the universal limit developed by Luby et al. (Luby, Sinclair, & Zuckerman 1993). The heuristic is a randomized version of the customized SetTimes heuristic used above. Again, the set of activities with minimum start time are selected. One activity from this set is randomly chosen by a biased roulette wheel weighted by the ratio of activity weight to duration. Higher weight, lower duration activities have a higher probability of being selected.

Two sets of $10 \times 10$ JSP problems are used. Initially 10 makespan instances were generated with an existing generator (Watson et al. 1999). The machine routings were randomly generated and the durations were uniformly drawn from $[1, 99]$. These instances were transformed into two sets of total weighted completion time problems with the only difference being the range of job weights: the first set has job weights uniformly drawn from the interval $[1, 9]$ and the second set has job weights uniformly drawn from the interval $[1, 99]$.

The models and algorithms were implemented in ILOG Scheduler 6.3. Experiments were run on 2GHz Dual Core AMD Opteron 270 with 2Gb RAM running Red Hat Enterprise Linux 4. We used an overall time-out of 1200 CPU seconds for each run. The randomized restart results are the mean over 10 independent runs.

Results

For this experiment, the dependent variable is mean relative error (MRE) relative to the best solution known for the problem instance. The MRE is the arithmetic mean of the relative error over each run of each problem instance:

$$MRE(a, K, R) = \frac{1}{|R||K|} \sum_{r \in R} \sum_{k \in K} \frac{c(a, k, r) - c^*(k)}{c^*(k)}$$

(1)

where $K$ is a set of problem instances, $R$ is a set of independent runs with different random seeds, $c(a, k, r)$ is the lowest cost found by algorithm $a$ on instance $k$ in run $r$, and $c^*(k)$ is the lowest cost known for $k$. As these problem instances were generated for this experiment, the best-known solution was found either by the algorithms tested here or by variations used in preliminary experiments.

Figures 1 and 2 display the results for the two problem sets. There are two main comparisons of interest:

1. COMPLETION vs. WS back propagation, with the search held constant. With either search technique and on both problem sets, the COMPLETION constraint with busy weight allocation (COMP-BUSY) significantly out-performs the other three variations (WS-BUSY, WS-LAST, and COMP-LAST). The difference is larger with chronological backtracking than with randomized restart.

2. Randomized restart (RR) vs. chronological backtracking (Chron) with the propagation held constant. The chronological search significantly out-performs randomized restart. For the problems with narrower weight range, even chronological backtracking with the weaker propagation out-performs randomized restart with the stronger propagation.

Generalizing to Discrete Resources

Extending the COMPLETION constraint to a discrete resource, we introduce the global constraint \(\text{COMPLETION}_m\), which states that given a set of non-preemptive activities \(\{A_1, \ldots, A_n\}\) that require the same discrete resource with capacity \(R\), the total weighted completion time of these activities is \(C\). The constraint takes the form:
where finite-domain variables \( S_i \) stand for the start time of \( A_i \), while \( p_i, q_i \), and \( w_i \) denote the duration, the capacity requirement, and the weight of \( A_i \), respectively. The cost variable \( C \) is also a finite-domain variable in the CSP. We assume that \( p_i, q_i, w_i \), and \( R \) are non-negative integer constants, however our approach can be easily adapted to reasoning with the lower bounds of \( p_i, q_i, \) and \( w_i \), and the upper bound of \( R \).

The minimum and maximum values in the current domain of a variable \( X \) will be denoted by \( X \) and \( \bar{X} \), respectively. When appropriate, we call the current lower bound of a start time variable, \( S_i \), the release time of the activity, and denote it by \( r_i \). For brevity, we denote the relative weight of an activity by \( \mu_i = w_i / p_i q_i \).

### The Relaxed Problem

Effective propagation requires embedding into the constraint a polynomially solvable relaxation of the single discrete resource total weighted completion time problem that considers capacity constraints, resource requirements, and release times at the same time. As such a relaxation does not appear to have been presented in the literature, below we propose a novel relaxation of this scheduling problem, see Fig.3.

Minimize

\[
\sum_{i,t} t \mu_i x^i_t
\]  

s.t.

\[
\forall i, t \quad x^i_t \geq 0
\]

\[
\forall i \quad \sum_t x^i_t = p_i q_i
\]

\[
\forall t \quad \sum_i x^i_t \leq R
\]

\[
\forall i, t \quad \sum_{t'=0} t \leq \begin{cases} 0 & \text{if } t < r_i \\ (t-r_i+1) q_i & \text{if } r_i \leq t < r_i + p \\ p_i q_i & \text{if } t \geq r_i + p \end{cases}
\]

Figure 3: The variable-intensity relaxation of the discrete resource scheduling problem.

In the relaxed problem, we assume that activities \( A_i \) can be executed with a varying intensity over time. That is, in each time period \( [t, t+1) \), \( t = 0, \ldots , T - 1 \), an intensity \( x^i_t \) of \( A_i \) can be chosen from \( [0, R] \). As an extremity of varying intensity, preemption is also allowed. The sum of the intensities over time has to match the original volume of the activity (4), and the capacity constraint must be respected (5). The release time constraint now states that \( A_i \) cannot be processed before \( r_i \) and its volume is released gradually afterwards (6). The objective is to minimize the total weighted mean busy time of the activities. In the sequel, we differentiate between the original problem and the variable-intensity relaxation by denoting the first as \( \Pi \), and the second as \( \Pi' \). \( C' \) will stand for the cost of the optimal relaxed solution.

**Proposition 1** \( C' + \frac{1}{2} \sum_i w_i (p_i + 1) \) is a valid lower bound on the original problem \( \Pi \).

**Proof:** The optimal solution of \( \Pi \) with cost \( C^* \) is a feasible solution of \( \Pi' \) as well, and it total weighted mean busy time is \( C^* - \frac{1}{2} \sum_i w_i (p_i + 1) \geq C' \).

The optimal solution of the variable-intensity mean busy time relaxation can be computed using the following procedure, called \( \text{PrepareRelaxed}(\cdot) \), which constructs the schedule chronologically. At each point in time \( t \) when a scheduling decision has to be made, the algorithm assigns intensities to activities in the order of non-increasing \( \mu_i \). The intensity of \( A_t \) will be the minimum of

- the volume of \( A_t \) that has already been released but not yet processed, \( \min(p_i q_i, (t-r_i+1) q_i - \sum_{t'=0}^{t-1} x^i_{t'}) \)
- the remaining capacity for the subsequent time period.

These intensity values are applied until the scheduled volume of an activity exceeds its released volume or the release time of another activity is reached. The algorithm finishes when all activities are completely processed. Since the number of scheduling decisions is at most \( O(n^2) \) and intensities can be assigned in \( O(n) \) time, the overall time complexity of the algorithm is \( O(n^3) \), independently of the length of the scheduling horizon.

**Proposition 2** The above algorithm builds an optimal schedule for the variable-intensity mean busy time problem.

**Proof:** Let \( \sigma \) be an arbitrary feasible schedule that differs from schedule \( \sigma^* \) built by our algorithm, such that the difference cannot be characterized by an interchange of intensities between activities with identical relative weights. Let \( t^1 \) be the earliest point in time and \( A_{t^1} \) be the activity with the highest \( \mu_{t^1} \) such that \( x^1_{t^1}[\sigma] \neq x^1_{t^1}[\sigma^*] \). The construction of the algorithm ensures that \( x^1_{t^1}[\sigma] < x^1_{t^1}[\sigma^*] \). Then, there exist a time \( t_2 \) and activity \( A_{t_2} \) with \( t_1 < t_2 \) and \( \mu_{t_2} > \mu_{t_1} \) such that increasing \( x^1_{t_2}[\sigma] \) and \( x^2_{t_2}[\sigma] \) and decreasing \( x^1_{t_1}[\sigma] \) and \( x^2_{t_1}[\sigma] \) preserves feasibility and improves the objective value. Therefore a schedule that differs essentially from the one built by the algorithm cannot be optimal.

Observe that the above procedure can easily be modified to \( \text{PrepareRelaxed}(A_i, \cdot) \), which computes optimal relaxed solutions for restricted problems where the start time of activity \( A_i \) is bound to \( t \). This can be achieved by assigning \( r_i = t \) and \( \mu_i = \infty \), which gives activity \( A_i \) the largest relative weight among all the activities and ensures that it starts at \( t \) and processed at intensity \( q_i \) throughout its duration.

### From Relaxed Solutions to Bounds Tightening

Below we propose algorithms that tighten the bounds of the start time variable domains by exploiting the above presented polynomially solvable relaxation. Similarly to the unary resource \text{COMPLETION} constraint, propagation is
based on computing (or approximating) the cost of the optimal relaxed solutions for restricted problems where an activity \( A_i \) must start at time \( t \). This restricted relaxed problem will be denoted by \( \Pi'(S_i, t) \), and the value of its optimal solution by \( C'(S_i, t) \). Formally, we exploit the following proposition:

**Proposition 3** If \( C'(S_i, t) > \hat{C} \), then \( t \) can be removed from the domain of \( S_i \).

However, in contrast to the unary case, we are not able to define efficient recomputation methods that determine in a low-degree polynomial time all restricted relaxed solution costs from one relaxed solution computed by \( \text{PrepareRelaxed} \). Instead, we apply the \( \text{PrepareRelaxed} \) procedure to compute two relaxed solutions for each activity, one for \( \Pi'(S_i = \hat{S}_i) \) and another for \( \Pi'(S_i = \bar{S}_i) \). If either of these relaxed solutions violate the current upper bound on the cost, then we estimate how the current lower/upper bounds of \( S_i \) have to be modified to achieve consistency. Since estimations are not exact, this procedure has to be iterated until bounds consistency is reached. The algorithm is presented in Figure 4, while the two different earliest and latest start time approximation methods are presented in detail afterwards.

**PROCEDURE** TightenBounds()

FORALL activity \( A_i \)

LOOP

\( \sigma := \text{PrepareRelaxed}(A_i, \hat{S}_i) \)

IF cost(\( \sigma \)) > \( \hat{C} \) THEN

\( \hat{S}_i' := \text{RecomputeEarliestStart}(\sigma, A_i) \)

LOOP

\( \sigma := \text{PrepareRelaxed}(A_i, \bar{S}_i) \)

IF cost(\( \sigma \)) > \( \hat{C} \) THEN

\( \bar{S}_i' := \text{RecomputeLatestStart}(\sigma, A_i) \)

Figure 4: Algorithm for tightening the bounds of the start time variables.

**Recomputing the Earliest Start Time** The earliest start time of activity \( A_i \) is adjusted based on the optimal relaxed solution \( \sigma \) for \( \Pi'(S_i = \hat{S}_i) \). In order to obtain lower bounds for \( \Pi'(S_i = t) \) with \( t > \hat{S}_i \), we introduce procedure \( \text{RecomputeEarliestStart}(\sigma, A_i) \) and a further relaxation as follows.

For the situation where \( A_i \) starts at \( t \), we consider a schedule \( \sigma \) in which intensities \( x_{ij} \) with \( t' < \hat{S}_i \) or \( t' \geq t + p_i \) equal the corresponding intensities in \( \sigma \). Activity \( A_i \) is processed from \( t \) until \( t + p_i \) at intensity \( \sigma_i \). Otherwise, in interval \([\hat{S}_i, t + p_i]\) we assume that the release times of all activities \( A_j \) with \( j \neq i \) equal \( \hat{S}_i \). Hence, those activities will be processed in non-increasing order of \( \mu_j \).

This further relaxation has two advantages: firstly, only a small section of the schedule, namely the section in interval \([\hat{S}_i, t + p_i]\) varies over different values of \( t \). Secondly, this section of the schedule can be represented as a queue. The queue, as well as the cost of the schedule, is updated incrementally for subsequent values of \( t \) at time \( O(n) \) for each step. This step is iterated until the cost of \( \sigma \) decreases below the current upper bound cost \( \hat{C} \). The earliest start time of \( A_i \) is then updated to this value of \( t \).

**Recomputing the Latest Start Time** Given an optimal relaxed solution \( \sigma \) with cost \( C' \) for \( \Pi'(S_i = \hat{S}_i) \), let \( t^* \) denote the earliest point in time with \( t^* \geq \hat{S}_i + p_i \) in this relaxed solution such that all the volume of the activities that has been released before \( t^* \) is processed before \( t^* \):

\[ \forall j \sum_{t=0}^{t^*} x_{ij}^t = \min(p_j \sigma_j, (t^* - r_j + 1) \sigma_j). \]

Furthermore, let \( W \) denote the total weight of activity fragments processed between \( \hat{S}_i \) and \( t^* \), including activity \( A_i \):

\[ W = \sum_{j=1}^{n} \sum_{t=\hat{S}_i}^{t^*} \mu_j x_{ij}^t. \]

Procedure \( \text{RecomputeLatestStart}(\sigma, A_i) \) computes these values \( t^* \) and \( W \), and adjust the latest start time of \( A_i \) according to the following proposition.

**Proposition 4** Activity \( A_i \) cannot start later than \( t = \hat{S}_i - \left[ \frac{C' - \hat{C}}{W} \right] \).

**Proof:** Consider a further relaxation of the relaxed problem where, in interval \([\hat{S}_i, t^*]\), resource capacity is increased from \( R \) to \( 2R \), and the released volume of each activity \( A_j \) is increased with \( \sum_{t'=\hat{S}_i}^{t^*} x_{ij}^t \). The optimal solution of this further relaxed problem is a schedule with all intensities in the interval \([\hat{S}_i, t^*]\) in \( \sigma \) moved earlier by \( (\hat{S}_i - t) \). The cost of this schedule is exactly \( C' - (\hat{S}_i - t) W \), from which the above proposition follows.

**Computational Experiments**

We ran computational experiments to test the efficiency of the proposed \( \text{COMPLETION}_m \) constraint on a set of single discrete resource scheduling problems with release times for minimizing total weighted completion time. We compared the performance of two models, one of them using weighted sum (WS) back propagation, the other using the \( \text{COMPLETION}_m \) constraint. The proposed propagation algorithms have been implemented in C++ and embedded into ILOG Solver and Scheduler versions 6.1. In the experiments, we applied the same adapted \( \text{SetTimes} \) branching heuristic with chronological backtracking as for the job shop case.

The problem instances were generated with a modified version of a previous benchmark generator for the single unary machine total weighted completion time problem (Pan 2007). The parameters of the generator are the number of activities, \( n \), which we took from \( \{15, 20, 25, 30\} \), the resource requirement range \( \alpha \in \{0.5, 1.0\} \), and the relative
release time range $\beta \in \{0, 0.2, 0.6, 1.0\}$. For each combination of the above values we generated 10 instances, which resulted in 320 different problem instances.

The capacity of the resource was fixed to $R = 10$. Activity durations $p_i$ were randomized from $[1, 100]$ with a discrete uniform distribution, weights $w_i$ from $[1, 10]$, and resource requirements $g_i$ from $[1, R]$. This leads to instances where approximately $k = \frac{2KR}{\alpha R + 1} \approx \frac{2}{R}$ activities are processed in parallel on the resource. Hence, the release times were randomized from $[0, 50.5\beta k/k]$. The experiments were run on a 1.86 GHz Pentium M computer with 1 GB of RAM, with a time limit of 600 seconds imposed.

The experimental results are presented in Table 1. Each row of the table contains combined results for given values of $n$ and $\beta$, achieved with the WS and the COMPLETION$_m$ models. The results do not depend significantly on the value of $\alpha$. For either of the models WS and COMPLETION$_m$, column $Opt$ displays the number of instances out of 20 that could be solved and the optimality of the solution has been proven. Columns Mean $RE$ and Max $RE$ contain the mean and maximum relative error compared to the best solutions known. Column $Nodes$ shows the average number of search nodes, while $Time$ presents the average search time, including the proof of optimality, or 600 seconds where the solver hit the time limit.

Problem instances with a low number of activities, $n = 15$, or with a high $\beta$ were solvable easily for both models, while instances with a greater $n$ and a lower $\beta$ often proved hard for both models. The COMPLETION$_m$ constraint reduced the number of search nodes for all the instances, often by two orders of magnitude. On the other hand, the computational effort invested in this pruning paid off only for the hard instances, i.e., for $n \geq 20$ and $\beta \leq 60$. For such instances, the COMPLETION$_m$ model found better solutions and proved optimality for more instances. Easier problems were solved more quickly with the simple weighted sum constraint.

From the low number of search nodes with the COMPLETION$_m$ constraint we conclude that the applied variable-intensity relaxation is sufficiently tight. On the other hand, the reason of the high computational cost was the high number of recomputation cycles within the $TightenBounds()$ procedure: on average, the recomputation of the earliest (latest) start times required 2–4 (4–10) cycles, while in a few extreme cases, up to 100 cycles were necessary to achieve consistency. This suggests that more accurate earliest and latest start time recomputation techniques are required. Further experiments are necessary to investigate whether a better trade-off between pruning strength and computational effort can be achieved by aborting recomputations before consistency is achieved. Also, the performance of the model with the COMPLETION$_m$ constraint has to be compared to a model in which the same relaxation is exploited in the form of a lower bound.

**Conclusions**

We investigated applications and extensions of the earlier defined COMPLETION constraint. In multiple-machine project scheduling problems, where activities linked by precedence constraints constitute jobs, weights and performance measures are often related to jobs. Since CP solution techniques infer over individual activities, the assignment of job weights to activities is a crucial issue. We defined a weight assignment heuristic, which allocates weights to the activities on the most loaded resource. For the criterion of total weighted completion time in job shop problems, we showed in computational experiments that the COMPLETION$_m$ constraint with this weight assignment outperforms standard representations of the cost function.

We introduced the COMPLETION$_m$ constraint for the total weighted completion time of activities on a discrete resource. The proposed propagation algorithms exploit a variable-intensity relaxation of the discrete-resource scheduling problem. The new constraint achieved significant pruning in single discrete resource scheduling problems, though, due to its high computational complexity, this did not always result in a reduction in overall solution time. Our future work will focus on the improvement of earliest/latest start time recomputation methods for the COMPLETION$_m$ constraint, and investigating the possibility of developing a generic framework for cost constraints in scheduling.

**Acknowledgments**

This research was supported in part by the Natural Sciences and Engineering Research Council, ILOG, S.A, the EU FP6 Net-WMS, and the NKFH 2010/2004 projects. A. Kovács acknowledges the support of the ERCIM Alain Bensoussan fellowship programme and the János Bolyai scholarship of the Hungarian Academy of Sciences.

**References**


Table 1: Experimental results: number of instances solved to optimality (Opt), mean and maximum relative error in percent (Mean/Max RE), average number of search nodes (Nodes) and average search time in seconds (Time) with the weighted sum (WS) constraint and with the COMPLETION$_m$ constraint.

<table>
<thead>
<tr>
<th>$r$</th>
<th>$\beta$</th>
<th>WS</th>
<th>COMPLETION$_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Opt</td>
<td>Mean RE</td>
</tr>
<tr>
<td>15</td>
<td>0.0</td>
<td>20</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>20</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.6</td>
<td>20</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>20</td>
<td>0.00</td>
</tr>
<tr>
<td>20</td>
<td>0.0</td>
<td>0</td>
<td>0.20</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>12</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>0.6</td>
<td>20</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>20</td>
<td>0.00</td>
</tr>
<tr>
<td>25</td>
<td>0.0</td>
<td>0</td>
<td>1.48</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>0</td>
<td>1.36</td>
</tr>
<tr>
<td></td>
<td>0.6</td>
<td>14</td>
<td>0.69</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>20</td>
<td>0.00</td>
</tr>
<tr>
<td>30</td>
<td>0.0</td>
<td>0</td>
<td>2.50</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>0</td>
<td>1.60</td>
</tr>
<tr>
<td></td>
<td>0.6</td>
<td>5</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>18</td>
<td>0.00</td>
</tr>
</tbody>
</table>


Iterative Improvement Strategies for Multi-Capacity Scheduling Problems

Angelo Oddi and Amedeo Cesta
ISTC-CNR
Rome, Italy
{name.surname}@istc.cnr.it

Nicola Policella
European Space Agency
Darmstadt, Germany
nicola.policella@esa.int

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

Abstract
Iterative Flattening Search (IFS) is an iterative improvement heuristic schema for solving scheduling problems with a makespan minimization objective. Given an initial solution, IFS iteratively applies two steps: (1) a subset of solving decisions are randomly retracted from a current solution (relaxation-step); (2) a new solution is incrementally recomputed (flattening-step). Since its introduction, several variations of IFS have been proposed over the original strategy. Such variations involve both different strategies for the relaxation step and for the incremental solving procedure. This paper investigates a missing point in the related literature and present initial results of a uniform study to evaluate the effectiveness of the “single component” strategies among those proposed till now. This paper introduces a framework to combine and experimentally evaluate different IFS strategies. Specifically, we examine the utility of: (i) operating with different relaxation strategies; (ii) using different strategies to built a new solution. We evaluate these extensions on benchmark instances of the Multi-Capacity Job-Shop Scheduling Problem (MCJSSP) also used in previous IFS studies. The experimental results shed light on the weaknesses and the strengths of the ideas proposed over the past years and suggest potentials for more effective IFS procedures.

Introduction
Iterative Flattening Search (IFS or iFLAT) (Cesta, Oddi, & Smith 2000) is an iterative improvement heuristic schema for solving scheduling problems with a makespan minimization objective. Given an initial solution, IFS iteratively applies two steps: (1) a subset of solving decisions are randomly retracted from a current solution (relaxation-step); (2) a new solution is incrementally recomputed (flattening-step). iFLAT performance was measured on a set of challenging Multi-Capacity Job-Shop Scheduling Problem (MCJSSP) instances. Since its original introduction, several variations have been proposed. Two works in particular (Michel & Van Hentenryck 2004; Godard, Laborie, & Nuitjen 2005) have extended the performance of the original strategy. Michel & Van Hentenryck (2004) identified an anomaly in iFLAT search and proposed a simple extension, which dramatically improved the quality of its schedules while preserving its computational efficiency. The key idea was to iterate the relaxation step multiple times, the resulting algorithm found many new upper bounds and produced solutions within 1% of the best upper bounds on average. Additional improvements were obtained in (Godard, Laborie, & Nuitjen 2005) with an approach which follows the same schema of iFLAT but uses different engines for the flattening and relaxation steps. Such a procedure was able to find additional optimal solutions and furtherly improved known upper-bounds for MCJSSP benchmarks.

The different IFS proposals involve both different strategies for the relaxation step and for the incremental solving procedure. However, till now, an uniform study to evaluate the effectiveness of the “single component” strategies were an open issue. To this purpose this paper proposes an uniform framework to combine and experimentally evaluate different IFS strategies. Specifically, we examine the utility of:

i) operating with different relaxation strategies, one targeted on removing decisions on the solution critical path and another one considering the whole solution;

ii) using different strategies to built a new solution, one posting precedence constraints among the activities and another one based on setting the start time of the activities.

We present here some initial experimental results that shed light on the relative weaknesses and strengths of the previously proposed strategies and start suggesting more effective and efficient IFS procedures.

The paper is organized as follows. We first review the iterative flattening search schema as evolved from previous work. The central part of the paper introduces the available relaxation and solving strategies and present a framework for comparing all of them. Next we recall the MCJSSP problem domain and benchmark problem sets used in our evaluation. Performance results are then given that demonstrate the leverage provided by the extended search procedures. We conclude by briefly discussing further opportunities to extend and enhance the basic iterative flattening search concept.

Problem Representation
Before describing the iterative flattening approach we introduce the modeling perspective on which this schema is based. According to this a scheduling problem is represented as a directed graph \(G(A, E)\). \(A\) is the set of activities
specified in MCJSSP, plus a dummy activity $a_{\text{source}}$ temporally constrained to occur before all others and a dummy activity $a_{\text{sink}}$ temporally constrained to occur after all others. $E$ is the set of precedence constraints defined between activities in $A$.

In the IFS search schema the reference representation of a solution $S$ is given as an extended graph $G_S$ of $G$, such that an additional set of precedence constraints is added to the original problem representation. This means that the set $E$ is partitioned in two subsets, $E = E_{\text{prob}} \cup E_{\text{post}}$, where $E_{\text{prob}}$ is the set of precedence constraints originating from the problem definition, and $E_{\text{post}}$ is the set of precedence constraints posed to resolve resource conflicts. In general the directed graph $G_S(A, E)$ represents a set of temporal solutions. The set $E_{\text{post}}$ is added in order to guarantee that at least one of those temporal solutions is also resource feasible.

In searching for a solution different search strategies apply. Following a Precedence Constraint Posting (PCP) approach the set of $E_{\text{post}}$ is naturally created reasoning on contention peaks. For example in (Cesta, Oddi, & Smith 1998; 2000; 2002) the precedences are selected by a basic Earliest Start Time Algorithm (ESTA). ESTA is designed to address more general, multi-capacity scheduling problems with generalized precedence relations between activities (i.e., corresponding to metric separation constraints with minimum and maximum time lags) but is used also in (Cesta, Oddi, & Smith 2000; Michel & Van Hentenryck 2004) to deal with multi-capacity resource contention peaks in MCJSSPs. Others use a different strategy, the decision to Set a Start Time (SST) of an activity, by imposing a rigid temporal constraint between the $a_{\text{source}}$ and the start-time of the interested activity. This strategy is very common in the scheduling literature, for example it is used in (Godard, Laborie, & Nuitjen 2005) and many others.

The original Iterative Flattening Search procedure (Cesta, Oddi, & Smith 2000) iterates two steps:

**Relaxation step:** a feasible schedule is relaxed into a possibly resource infeasible, but precedence feasible, schedule by removing some search decisions represented as precedence constraints between pair of activities;

**Flattening step:** a sufficient set of new precedence constraints is posted to re-establish a feasible schedule.

This schema integrates naturally with the graph representation $G_S$ for a solution, that is the solution representation used in a PCP approach. To apply the same philosophy with an SST we need to relax the “solution rigidity” introduced by the absolute temporal constraints inserted as decisions. One way of doing it consists in transforming the SST solution in a Partial Order Schedules (POS) (Cesta, Oddi, & Smith 1998; Policella et al. 2004).

In general a POS can be also obtained from a PCP solution produced by ESTA. Both $G_S$ and POS are graph representations. The difference is that while ESTA solutions guarantee that at least one of the temporal solutions they represent is also resource feasible, a POS guarantees that all delineated temporal solutions are also resource feasible. The use of a POS in general increases the possibilities for rearranging relaxed activities. This is why we dedicate a short paragraph to POS basics.

**Partial Order Schedules**

The common thread underlying a POS 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 that 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 flexible if and only if there is temporal slack that allows chained activities to move “back and forth”. Polynomial methods for producing a POS from an input solution represented as a precedence graph (or equivalently as a set of start times) have been introduced in (Cesta, Oddi, & Smith 1998; Policella et al. 2004). Given an input solution, a transformation method, named *Chaining*, is defined that proceeds to create sets of chains of activities. This operation can be accomplished over three steps: (1) all the previously posted leveling constraints are removed from the input partial order; (2) the activities are sorted by increasing activity earliest start times; (3) for each resource and for each activity $a_i$ (according to the increasing order of start times), one or more predecessors $a_j$ are chosen, which supplies the units of resource required by $a_i$ – a precedence constraint $(a_i, a_j)$ is posted for each predecessor $a_j$. The last step is iterated until all the activities are linked by precedence chains.

Having a flexible solution is not the only benefit in considering the use of partial order schedules. A second property that appears to be relevant is the reduction in the number of additional precedence constraints that must be posted to obtain a solution: given a problem with $n$ activities to be scheduled, the number of constraints appearing in the solution is always $O(n)$. This because the chaining procedure creates POSs with only the “necessary” precedence constraints, and eliminates all “redundant” constraints. For example applying POS to a $G_S$ result on a PCP version of iFLAT removes redundant precedence constraints and tends to intensify the effect of the iFLAT relaxation step in the procedure. More solutions are accessible at each flattening cycle, because the removal of redundant constraints increases possibilities for rearranging relaxed activities. In (Godard, Laborie, & Nuitjen 2005) a POS is created from an SST solution to insert temporal flexibility in the solution before performing a relaxation step.

**Iterative Flattening Search**

Given these preliminaries we introduce a general IFSEARCH procedure in Figure 1. The algorithm basically alternates Relaxation and Flattening steps until a better so-
IFSSEARCH$(S, MaxFail)$
begin
  1. $S_{best} \leftarrow S$
  2. counter $\leftarrow 0$
  3. while (counter $\leq$ MaxFail) do
    4. relax$(S)$
    5. Sol $\leftarrow$ FLATTEN$(S)$
    6. if Mk(Sol) $<$ Mk($S_{best}$) then
      7. $S_{best} \leftarrow S$
      8. counter $\leftarrow 0$
    else
      9. counter $\leftarrow$ counter + 1
  10. return ($S_{best}$)
end

Figure 1: The IFSSEARCH general schema

Relaxation Procedures
In general, a relaxation procedure transforms a feasible schedule into a possibly resource infeasible, but temporal feasible, schedule by adopting different strategies for removing some search decisions. We have reproduced two of these strategies. The first, introduced in the paper (Cesta, Oddi, & Smith 2000; Michel & Van Hentenryck 2004), removes precedence constraints between pair of activities on the critical path of the solution, hence we call it pc-based relaxation; the second, introduced in the work (Godard, Laborie, & Nuitjen 2005), which starting from a POS-form solution, basically randomly breaks some chains in the input POS schedule, hence the name chain-based relaxation.

Precedence relaxation. The relaxation step is based on the concept of critical path. A path in $G_S(A, E)$ is a sequence of activities $a_1, a_2, \ldots, a_k$, such that, $(a_i, a_{i+1}) \in E$ with $i = 1, 2, \ldots, (k - 1)$. The length of a path is the sum of the activities processing times and a critical path is a path from $a_{source}$ to $a_{sink}$, which determines the solution's makespan. Any improvement in makespan will necessarily require change to some subset of precedence constraints situated on the critical path, since these constraints collectively determine the solution's current makespan. Following this observation, the relaxation step introduced in (Cesta, Oddi, & Smith 2000) is designed to retract some number of posted precedence constraints on the solution's critical path. Figure 2 shows the PCRELAX procedure. Steps 2-4 consider the set of posted precedence constraints $(p_r \in E_{post})$, which belong to the current critical path. A subset of these constraints is randomly selected on the basis of the parameter $p_r$ $\in$ (0, 1) and then removed from the current solution. Step 1 represents the crucial difference between the approaches of (Cesta, Oddi, & Smith 2000) and (Michel & Van Hentenryck 2004). In the former approach Steps 2-4 are performed only once (i.e., $MaxRLxs = 1$), whereas in (Michel & Van Hentenryck 2004) these steps are iterated several times (from 2 to 6), such that, a new critical path of $S$ is computed at each iteration. Notice that this path can be completely different from the previous one. This allows the relaxation step to also take into account those paths that have a length very close to the one of the critical path.

Chain Relaxation. This second relaxation requires an input solution in POS-form. A solution in POS form is an extension of the original precedence graph representing the

PCRELAX$(S, p_r, MaxRLxs)$
begin
  1. for 1 to $MaxRLxs$
  2. forall $(a_i, a_j) \in$ CriticalPath$(S) \cap E_{post}$
  3. if random(0,1) $< p_r$ then
    4. $S \leftarrow S \setminus (a_i, a_j)$
end
input scheduling problem. As previously introduced, a POS
form solution is a graph \( G_S(N, E_{prob} \cup E_{ach}) \), such that the
set \( E = E_{prob} \cup E_{ach} \) is partitioned into a set of chains
\( CH_1, CH_2, \ldots, CH_{nc} \). Each chain \( CH_i \) imposes a total
order on the subset of problem activities requiring the same
resource. Hence, given a generic activity \( a_k \), \( \text{pred}(a_k) = \{a_p | \exists CH : (a_p, a_k) \in CH \} \) is the set of its predeces-
sor activities and \( \text{succ}(a_k) = \{a_s | \exists CH : (a_k, a_s) \in CH \} \) is the set of its successors activities. In particular,
\( \text{pred}(a_{source}) = \text{succ}(a_{sink}) = \emptyset \) Figure 3 shows the
chain-based relaxation procedure. The procedure (a) ra-
domly selects a subset of activities from the input solution
\( S \) on the basis of the parameter \( p_r \in (0, 1) \), (b) removes the
edges \( (a_p, a_k) \), \( a_p \in \text{pred}(a_k) \) and \( (a_k, a_s) \), \( a_s \in \text{succ}(a_k) \)
without updating the start times \( \text{est} \) of the activities; (c) the
\textit{Chaining} procedure (previously described) is applied on the
set of unselected activities, that is, the activities not removed
by the random selection. It is worth observing that such
activities still represents a feasible solution to a schedul-
ing sub-problem, which can be transformed in POS-form,
in which the randomly selected activities float outside the
solution thus re-creating contention peaks.

Flattening Procedures

Both relaxation schema create a solution with contention
peaks that should be flattened. We have implemented two
general solution schema, one based on the PCP idea, the sec-
ton on the SST strategy. Both solving algorithms are able to
perform a complete search through backtracking.

PCP Search (PCPS). The flattening step (see Figure 4) used in (Cesta, Oddi, & Smith 2000) is inspired by prior
work on the Earliest Start Time Algorithm (ESTA) from
(Cesta, Oddi, & Smith 1998). The algorithm is a variant
of a class of PCP scheduling procedures characterized by a
two-phase solution generation process. The first step con-
structs an infinite capacity solution. The current problem
is formulated as an STP (Dechter, Meiri, & Pearl 1991)
temporal constraint network\(^3\) where temporal constraints are

\[ \text{PCPS}(P, S) \begin{align*}
\text{begin} & \quad \text{1. Propagate}(S) \\
\text{if IsSolution}(S) & \quad \text{3. then return}(S) \\
\text{else} & \quad \text{4.} \\
\text{mc} & \quad \text{5. mcs} \leftarrow \text{SelectConflict}(P, S) \\
\text{if Solvable}(mcs, S) & \quad \text{6. if} \\
\text{then} & \quad \text{7.} \\
\text{pc} & \quad \text{8. \quad pc} \leftarrow \text{ChoosePrecedence}(S, mcs) \\
\text{MCSS}(P, S \cup \{pc\}) & \quad \text{9.} \\
\text{else return(fail)} & \quad \text{10.} \\
\end{align*} \]

Figure 4: The PCPS algorithm

\(^3\)In a STP (Simple Temporal Problem) network we make the
following representational assumptions: temporal variables (or
modeled and satisfied (via constraint propagation) but re-
source constraints are ignored, yielding a time feasible solu-
tion that assumes infinite resource capacity. The second step
levels resource demand by posting precedence constraints.
Resource constraints are super-imposed by projecting “re-
source demand profiles” over time. Detected resource con-
flicts, which are Minimal Conflict Sets (MCS) as in (Cesta,
Oddi, & Smith 2002), are then resolved by iteratively post-
ing simple precedence constraints between pairs of compet-
ing activities. The constraint posting process of ESTA is based on the Earliest Start Solution (ESS) consistent with currently
imposed temporal constraints. It then proceeds to compute a resource conflict (Step 2-5). If this set is empty
the ESS is also resource feasible and a solution is found;
otherwise if a conflict exists that can be solved, a new prece-
dence constraint is posted to do so (Steps 8-9); otherwise the
process fails (Step 10). For further details on the functions
\textit{SelectConflict(), and ChoosePrecedence()} (non determinis-
tic version of the precedence selection operator) the reader
should refer to the original references.

SST Search (SSTS). The second solving procedure is based on the idea of searching the set of possible assignments
to the activity start-times. In particular, our implemen-
tation of SSTS can be seen as a serial scheduling schema (Kolisch 1996) adopting the latest finish time (LFT)
priority rule, which branches the search on the possible ear-
liest start times (Dormdorf, Pesch, & Phan Huy 2000). How-
ever, other search schemas are possible, with different prior-
ity rules, this will be motivation for further experiments in
the near future. A recursive and non deterministic version
of the solver is shown in Figure 5. At Step 1 the procedure
\textit{Propagate} propagates the current temporal constraints.
In particular, for each activity \( a_i \) updates its earliest start-time \( \text{est}_i \) and latest finish time \( \text{LFT}_i \) of the activities. When the
output solution \( S \) is a complete and resource feasible solu-
tion (all the activities has a start-time assigned), the pro-
cedure returns it (Steps 2-3). Otherwise an activity is selected
on the basis of a priority rule. Currently, we select the ac-
tivity with the minimal latest finish time \( \text{LFT} \) (ties are broken by the \( \text{est} \) values). Given a selected activity \( a_i \), the search
branches (Step 8) on the possible resource feasible assign-
ments of the earliest start-time \( \text{est}_i \).

Iterative Flattening Variants

The concept of Iterative Flattening introduced in (Cesta,
Oddi, & Smith 2000) is quite general and provided an in-
teresting new basis for designing more sophisticated and
effective local search procedures for scheduling optimiza-
tion. The \textit{iFLATRELAX} procedure proposed in (Michel &
Van Hentenryck 2004) is a nice example of an \textit{iFLAT} exten-
sion which obtains substantial improvements over its origi-
nal version. In addition, the version of Iterative Flattening
proposed in (Godard, Laborie, & Nuitjen 2005) produced
further improvements on both the previous procedures. This
variants allow for the representation of activities as a set
(time-points) represent the start and end of each activity, and the
beginning and end of the overall temporal horizon; distance con-
straints represent the duration of each activity and separation con-
straints between activities including simple precedences.
procedure uses a solving strategy similar to SSTS and a chain-based relaxation schema.\(^4\)

However, till now, an uniform study to evaluate the effectiveness of the single component strategies proposed in the literature were an open issue. In this spirit, the paper proposes an uniform framework to combine and experimentally evaluate different IFS strategies. Specifically, we examine the utility of: (i) operating with different relaxation strategies; (ii) using different strategies to built a new solution. Hence, our idea is to shed light on the weaknesses and the strengths of the ideas proposed over the past years and suggest more effective and efficient IFS procedures. According to this idea we propose the following IFS procedures:

- Two procedures based on PCPS search, one uses the precedence relaxation – identified with PCs – and another one the chain relaxation - identified with ACTs. PCPS is implemented as a depth-first backtracking procedure using an input parameter α, which is used to limit the number of backtracking steps. In particular, the PCPS procedure returns the solution found with minimal makespan, within αn steps, where n is the number of problem’s activities. We observe that the combination of PCPS and PC-based relaxation with α = 0 reproduces the algorithm in (Michel & Van Hentenryck 2004) furtherly extended with a backtracking search procedure.

- Two IFS procedures based on SSTS search, one with precedence relaxation – called SSTS-PCs – and another one with chain relaxation – called SSTS-ACTs. Also in this case, SSTS search uses the same parameter α to bound the number of backtracking steps to the value αn and returns the best solution found with regard to the makespan.

- A new IFS procedure – called PCPS-ACTs-iPCs – which coincides with the combination of PCPS and Chain-based relation, except when an improved solution is found within the Iterative Flattening loop (see Steps 3-10 in Figure 1). In this case the relaxation procedure is temporary switched to the precedence based one.

- A new IFS procedure – called SSTS-ACTs-iPCs – which mirrors the previous one on the relaxation strategy, but uses the SSTS search procedure.

As introduced above, the main goal of this paper is perform a first uniform study for evaluating the strengths and the weaknesses of the single IFS component strategies. In particular, the first four IFS strategies, basically combines already know procedures, even if two of them (PCPS-ACTs and SSTS-PCs) are relatively new algorithms. Whereas, the last two procedures, proposes two new algorithm based on the following intuition. We observe, the PC-based relaxation is more targeted on directly reducing the makespan of a solution, because specifically relaxes its critical path, which is directly correlated to the solution’s makespan. However, such procedure seems also more prone to be trapped in a local minima. On the contrary, the Chain-based relaxation removes activities independently from the critical path, hence it promotes a search with an higher degree of diversification. The last two IFS procedures are two attempts to interleave intensification and diversification mechanisms within the same IFS procedure in order to improve performance. In the next section, after a short summary on the used benchmarks, we propose a first empirical evaluation of the procedures defined in this section.

The MCJSSP Scheduling Problem

We consider the Multi-Capacity Job-Shop Scheduling Problem, MCJSSP, as a basis for evaluating the performance of our search procedures. This problem involves synchronizing the use of a set of resources \( R = \{ r_1 \ldots r_m \} \) to perform a set of jobs \( J = \{ j_1 \ldots j_n \} \) over time. The processing of a job \( j_i \) requires the execution of a sequence of \( m \) activities \( \{ a_{i1} \ldots a_{im} \} \), each \( a_{ij} \) has a constant processing time \( p_{ij} \) and requires the use of a single unit of resource \( r_{ai} \) for its entire duration. Each resource \( r_j \) is required only once in a job and can process at most \( c_{ij} \) activities at the same time (\( c_{ij} \geq 1 \)). A feasible solution to a MCJSSP is any temporally consistent assignment to the activities’ start times which does not violate resource capacity constraints. An optimal solution is a feasible solution with minimal overall duration or makespan. Generally speaking, MCJSSP has the same structure as JSSP but involves multi-capacitated resources instead of unit-capacity resources.

Benchmark Sets

For our analysis, we refer to the benchmarks introduced in (Nuijten & Aarts 1996). They consist of four sets of problems which are derived from the Lawrence job-shop scheduling problems (Lawrence 1984) by increasing the number of activities and the capacity of the resources.

Set A: \( LAI-10x2x3 \) (Lawrence’s problems numbered 1 to 10, with resource capacity duplicated and triplicated). Using the notation \#jobs \times \#resources (resource capacity), this set consists of 5 problems of sizes 20x5(2), 30x5(3), 30x5(2), 45x5(3).
Set B: LA11-20x2x3. 5 problems each of sizes 40x5(2), 60x5(3), 20x10(2), 30x10(3).
Set C: LA21-30x2x3. 5 problems each of sizes 30x10(2), 45x10(3), 40x10(2), 60x10(3).
Set D: LA31-40x2x3. 5 problems each of sizes 60x10(2), 90x10(3), 30x15(2), 45x15(3).

We observe that the proposed benchmark set still represents a challenging benchmark for comparing algorithms. In fact, (a) in relatively few instances they cover a wide range of problem sizes; (b) they also provide a direct basis for comparative evaluation. In fact, as noted in (Nuijten & Aarts 1996), one consequence of the problem generation method is that the optimal makespan for the original JSSP is also a tight upper bound for the corresponding MCJSSP (Lawrence upper bounds). Hence, even if for many instances there are known better solutions, distance from these upper-bound solutions can provide a useful measure of solution quality.

Current Experimental Results
This section proposes a first explorative evaluations of the IFS procedures introduced in the previous sections. In this phase of our work, we are using the Set C benchmark, which is a quite representative sub-set of the proposed full benchmark of MCJSSP instances. It contains very interesting instances ranging from 300 to 600 activities and is really suitable for exploring interesting trends before a time consuming intensive testing. All algorithms were implemented in Allegro Common Lisp and were run on a Pentium 3 processor 800 MHz, under Windows XP.

The general settings for the tested IFS strategies were the following:
1. we have limited the amount of backtracking for the procedures PCPS and SSTS by setting $\alpha = 2$;
2. the parameters for the precedence-based relaxation were $p_r = 0.2$ and $MaxFail = 6$;
3. the parameter $p_r$ of the chain-based relaxation was set to 0.1 and 0.2;
4. we imposed a timeout of 3200 seconds for each problem instance and for each strategy we set $MaxFail = 1600$ (the maximum number of non improving moves that the algorithm tolerates before termination).

In addition, in order to meet the imposed timeout, we adopt the same restarting schema used in previous works (Cesta, Oddi, & Smith 2000; Michel & Van Hentenryck 2004). In the case a first run finishes before the imposed time limit, the random procedure restarts from the initial solution until the time bound is reached. At the end, the best solution found is returned.

Table 1 compares the performance of the IFS strategies with respect to the value $\Delta LWU_{\%}$, which represents the average percentage deviation from the Lawrence upper bound (Lawrence 1984). In particular, given a numeric value in the table, (for example 9.84) the corresponding IFS strategy is given by reading the column’s label (PCPS or SSTS), representing the solving strategy, and the row’s label (one among PCs, ACTs or ACTs-iPCs) representing the adopted relaxation strategy. In particular:

- PCs row represents the precedence-based relaxation on the solution critical path,
- ACTs represents the chain-based relaxation,
- ACTs-iPCs represents the chain-based relaxation with the switching to the PCs relaxation when the makespan improves within the iterative flattening loop.

Some of the relaxation strategies are differentiated with respect to the value of the parameter $p_r$ (the probability to randomly remove an activity in a POS-form solution). Hence, the value 9.84 in Table 1 refers to an IFS algorithm using SSTS search and the relaxation strategy ACTs-iPCs with $p_r = 10$. The remaining IFS procedures can be easily deduced in analogous way.

First of all, the results shown in Table 1 gives a first empirical evidence of the fact that within the same computational framework, PCPS search performs better than SSTS. We remember our implementation of SSTS can be seen as a serial scheduling schema adopting the latest finish time (LFT) priority rule, which branches the search on the possible earliest start times. Other search schemas are possible, with different priority rules, this will be investigated stimulus in the near future.

When we consider the first three rows of Table 1, we clearly see that ACTs always outperforms PCs. In particular, the best performance is obtained by the combination of PCPS and ACTs. A possible explanation of this fact is that precedence-based relaxation is more targeted on directly reducing the makespan of a solution, because specifically relaxes its critical path (which is directly correlated to the solution’s makespan). Hence, such procedure seems also more suited procedure to explore the detected diversification thus explaining the better performance observed.

Things get even more interesting when we read also the last two rows of Table 1, where we see that the IFS procedures using PCPS and the relaxation strategy ACTs-iPCs improves over the other PCPS-based procedures. Notice that the last two IFS procedures are a first attempt to interleave intensification and diversification within the same IFS procedure. In particular, the idea is that when an improvement of the makespan is detected within the IFS loop, the relaxation strategy is temporary switched to the PCs one, which should be the more suited procedure to explore the detected
local minima. When no more improvement is found, the relaxation strategy is restored back to the ACSs one, which promotes a search with an higher degree of diversification.

A last comment concerns the parameter $p_r$, representing the probability of removing at random an activity in a POS-form solution. Here we consider two different values ($0.1$ and $0.2$) just to test the sensibility of the performance measure with respect to $p_r$. Again, we remark the need for a more in-depth experimentation. Nevertheless, we observe the opposite effect with respect to the IFS flattening procedures. In fact, the best performance for the PCPS-based procedures is obtained with $p_r = 0.1$, whereas the best performance for the SSTS-based ones are obtained with $p_r = 0.2$.

Conclusions

In this paper we have discussed a set of extensions to the Iterative Flattening Search procedure. IFLAT is a local search procedure for solving large-scale scheduling problems with a makespan minimization objective criterion. The presented extensions were motivated to perform an uniform study to evaluate the effectiveness of the “single component” IFS strategies proposed in the literature. In this spirit, we propose an uniform framework to combine and experimentally evaluate different IFS strategies. Specifically, we examine the utility of:

i) operating with different relaxation strategies, one targeted on removing decisions on the solution critical path and another one considering the whole solution;

ii) using different strategies to built a new solution, one post-ing precedence constraints among the activities and another one based on setting the start time of the activities.

We proposed a first experimental evaluation on benchmark instances of the Multi-Capacity Job-Shop Scheduling Problem, which have been used in previous studies of IFS procedures. The present experimental results start to clarify some weaknesses and strengths of the ideas proposed over the past years and suggest more effective and efficient IFS procedures. Some of the proposed extensions were found to improve the performance of the reference strategies. We are planning now to start an intensive experimentation on the complete set of MCISSP benchmarks.

Acknowledgments

Amedeo Cesta, and Angelo Oddi’s work is partially supported by MIUR (Italian Ministry for Education, University and Research) under the project VINCOLI E PREFERENZE (PRIN). Nicola Policella is currently supported by a Research Fellowship of the European Space Agency, Directorate of Operations and Infrastructure. Stephen F. Smith’s work is supported in part by the National Science Foundation under contract #9900298, by the Department of Defense Advanced Research Projects Agency under contract # FA8750-05-C-0033 and by the CMU Robotics Institute.

References


Planning and Scheduling Teams of Skilled Workers

Laurent Perron, Paul Shaw, Didier Vidal
ILOG SA, 9 rue de Verdun, 92453 Gentilly Cedex, France

Abstract
Solving problems that mix planning and scheduling are often seen as a challenge. Discrete time-based scheduling, along with complex side constraints does not mix well with the more flexible nature of the planning model. This is demonstrated in our experiments when trying to solve a problem where we must assemble teams of skilled workers to perform jobs that require these skills, break these teams and then assemble new ones to perform more jobs. The mixing of the planning part (grouping workers into teams) and the scheduling part (creating a schedule for each worker), along with some difficult side constraints and a large problem size (800 workers, 2000 jobs over one month) combine to contribute to the challenge of finding good solutions for this problem.

Introduction
Planning and scheduling, the juxtaposition of the two names stems from the technical limitations of the engines used to solve them. On the one hand, we deal with the approximated nature of the long term planning; and we often use math programming to solve it. On the other hand, the discretization or bucketization of time, the low-level side constraints, the special cases and requests that have been approximated out in the planning phase, all ask for another kind of solver, often a constraint-based one.

In fact, we would like to get rid of the distinction and solve both problems at once. This is like solving the crew pairing and the crew scheduling problem at the same time in the airline industry, or the capacity planning and the detailed scheduling in the same model for the discrete manufacturing world.

However in doing so, we often face all kind of difficulties from fitting the model in memory to finding feasible solutions, even trivial ones as the solver has to deal with a heterogeneous model, in which search guidance information becomes lost or difficult to extract.

This article tells a version of the same story. The complex and heterogeneous nature of a timetabling problem forced us to look at a decomposition to get a grip on the problem itself. We tried different methods of avoiding a decomposition, from complex modeling to heuristics to reduce the problem size and complexity. All techniques were pitted and evaluated against a simple decomposition schema were the linear constraints were separated from the scheduling ones and given respectively to a MIP solver and a CP solver.

We began our work with an interesting timetabling problem with some twists: travel constraints, set covering constraints, knapsack constraints. We looked at it and came up with two alternative models for it. Both were evaluated against tiny, small, medium and large data sets and the results were extremely disappointing as one was able to treat only the tiny problems and the other was able to treat the tiny and the small ones. However, the goal was to solve the large instances. We were far from success at that time.

To deal with the size of the largest models, we tried two approaches. The first one was to give the packing and set covering part to ILOG CPLEX(CPLEX 2007) and the rest to ILOG CP Optimizer(CP-Optimizer 2007).

The second one was to do some something like simple column generation where part of the problems were pre-computed (the packing + set covering part). The the master problem was not a linear one but a timetabling one and was solved with ILOG CP Optimizer.

The article is divided in four sections. The first one will present the problem and discuss its nature. It will also present the implementation details of the side constraints. The second section will present our initial failed experiments. The third section speaks about decomposition and model improvements. The last section presents experimental results on the final two approaches.

Presentation of the Problem
The skilled team problem can be described as follows. Given a set of skills like painting, plumbing, roofing, a set of workers with these skills and a set of jobs that requires these skills, the goal is to assign workers to jobs and to find a start date for each job such that they form an acceptable schedule for each worker. Meaning, all workers participating in the same job work on the same days. A worker can perform at most one job per day. And finally, if a worker has to go to a distant (far) job, he must stay at a hotel before and after
this job. In addition, if a worker returns home because he has no job that day, then he cannot leave the same day. This is equivalent to saying that there cannot be exactly one free day between two jobs which are far from home.

This model is a closely related to the audit scheduling problem (Balachandran & Zoltners 1981; Chan & Dodin 1986; J.C. & Lofti 1990; Dodin 1991). Different methods have been proposed to solve it (Bajis & Elimam 1996; Dodin, Elimam, & Rolland 1996; Drexl, Frahm, & Salewski).)

**Model Description**

Given a set of Location $L = \{ l_1, \ldots, l_n \}$ along with a decision procedure $\text{bool} \ far(l_i, l_j)$.

Given a set of Skills $S = \{ s_1, \ldots, s_{#s} \}$.

Given a set of Jobs $J = \{ j_1, \ldots, j_{#j} \}$.

where $j_i = < l, d, n, s \subseteq S, w >$ with $l$ the index of the location of the job, $d$ the number of days needed to perform the job, $n$ the number of workers needed for the job, $s$ the subset of $S$ of skills required by the job and $w$ the weight (importance) of the job.

Given a set of Workers $W = \{ w_1, \ldots, w_{#w} \}$.

where $w_i = < l, s \subseteq S >$ where $l$ is the index of the location of the home of the worker and $s$ is the set of skills the worker is qualified for.

Given a number of work days $nd$.

Given the following variables:

$\text{bool} \ a_{x,y}; x \in [1..#w], y \in [1..#j]$ $w_x$ performs $j_y$

$\text{bool} \ b_y; y \in [1..#j]$ $j_y$ is performed

$\text{int} \ t_y \ in \ [0..nd]; y \ in [1..#j]$ start time of $j_y$

The problem can be stated as:

$$\max \sum_{y \in 1..#j} J_y.w \times b_y$$

subject to

- **card:** $\forall x \sum_y a_{x,y} = b_y \times j_y.n$
- **day worked:** $\forall y \sum_x a_{x,y} \times j_y.d \leq nd$
- **skill covering:** $\forall y \sum_{x,y} a_{x,y} \otimes w.s \geq j_y.s \otimes b_y$
- **unperformed:** $\forall y t_y = 0 \Leftrightarrow b_y = \text{false}$
- **valid schedule:** At most one job at a time per worker
- **home:** If idle, a worker is at home
- **forbidden:** Far/home/far is forbidden

In the above model, $s \otimes b$ with $s$ a set and $b$ a boolean value is defined as $\emptyset$ if $b$ is $\text{false}$ and $s$ if $b$ is $\text{true}$.

**Discussion**

In this problem, we can distinguish between three subproblems. The first one is a constrained variation on the knapsack problem where we want to pack jobs to workers and maximize the pack value. The second one is a set covering problem to determine valid combination of workers to assign to a particular job. The last one is derived from a classing scheduling problem with alternative resources and some specific forbidden transitions between activities.

An important aspect of this problem is the size of it. The real life problem this model is derived from counts 800 workers, 2000 jobs, fifteen skills and the scheduler spans roughly twenty days.

Therefore, we have to be careful about model complexity. Let’s imagine we maintain a precise agenda for each worker featuring the exact job he is performing each day. Then implementing the compatibility table that will link three consecutive days has a size of $2000 \times 2000 \times 800 = 3.2$ billion cells in the dense graph of the relation!

Thus implementing the precise constraint cannot be done in a naive way. This will be the subject of the next section.

**Implementation of the Home and Forbidden constraints as a Disjuction**

As seen in the previous section, the tricky part in the implementation of the model is the definition of a valid schedule that will express correctly the forbidden sequence constraint.

We first tried to implement the complete schedule with just the start variables $(t_y)$ of the jobs. In that case, we can add the following constraint to state the natural disjunction between jobs than can be performed or not:

$$\text{disjunct1: } \forall_{x,y,y'} a_{x,y} \land a_{x,y'} \Rightarrow (t_y \geq t_{y'} + j_{y'.d}) \lor (t_y \geq t_{y'} + j_{y'.d})$$

Of course, this formulation is quadratic, and thus does not scale well. Even if we filter out $a_{x,y}$ that we can safely set to false there remains a huge number of constraints of this type.

In addition, this type of constraint (disjunction) is typically handled less efficiently than global or specialized constraints in typical CP solvers. Regardless of the quadratic complexity, we will try to improve the individual constraints themselves.

A better formulation would be to replace the implication by a term in the sum that would nullify the constraint if $a_{x,y} \land a_{x,y'}$ is false. This formulation is a bit better as it allows slightly more propagation.

$$\text{disjunct2: } \forall_{x,y,y'} (t_y \geq t_{y'} + j_{y'.d} + \alpha_{x,y,y'}) \lor (t_y \geq t_{y'} + j_{y'.d} + \alpha_{x,y,y'})$$

$$\text{interact: } \alpha_{x,y,y'} = (a_{x,y} \times a_{x,y'} - 1) \times M$$

where $M$ is a big enough constant and $\alpha_{x,y,y'}$ is a three dimensional array on intermediate expressions. This kind of formulation is common in the math programming community.

Using this type of formulations, we can add a constraint a simple constraint that is stronger than the forbidden constraint stating that if two jobs are far from the same worker and can interact, then they cannot be one day apart.

---

1Because $w_x, s \cap j_{y,s} = \emptyset$.

2greater than $nd$ for instance.

3That are lazily generated, in order not to hit the dreaded $\#j \times \#j \times \#w$ complexity.
This constraint is actually cutting valid solution as it would have been possible to have a one day job in between two far jobs. We will evaluate them in the experimentation section.

Maintaining the Precise Agenda of Workers

Another possible implementation is to introduce variables that will record the precise agenda of workers.

\[
\text{int } g_{x,d} \text{ in } [0..\#j]; x \in [1..\#w], d \in [1..\#nd]
\]

The variable \(g_{x,d}\) represent the job performed by the worker at the date \(d\). A value of zero indicates that the worker is idle.

To help implement the \textit{forbidden} and \textit{home} constraints, we will introduce three sets of auxiliary variables:

\[
\begin{align*}
\text{bool } h_{x,d} &; x \in [1..\#w], d \in [1..\#nd] \\
\text{bool } f_{x,d} &; x \in [1..\#w], d \in [1..\#nd] \\
\text{int } \text{worked}_{x} &; x \in [0..\#w]
\end{align*}
\]

where \(h_{x,d}\) is \textit{true} when the worker \(x\) is idle on day \(d\), \textit{false} otherwise; and \(f_{x,d}\) is \textit{true} when the worker \(x\) is working far from home on day \(d\) and \textit{false} otherwise. The variable \textit{worked}_{x} computes the total number of days worked per workers.

When this is done, we can pose constraints that will set the \(g\) and \(f\) variables when a job is assigned to a worker.

\[
\begin{align*}
\text{agenda: } & \forall x,d, a_{x,y} \Rightarrow \bigwedge_{\delta \in [0..j_{y}.d-1]} g_{x,t_{y}+\delta} = y \\
\text{far1: } & \forall x,d, a_{x,y} \Rightarrow \\
& \bigwedge_{\delta \in [0..j_{y}.d-1]} f_{x,t_{y}+\delta} = \text{far}(w_{x}.l, j_{y}.l)
\end{align*}
\]

Computing the \(h\) variables is a bit more complex. As the constraints that maintain the agendas are implications between the \(a\) variables and the \(g\) variables, deciding if a worker is idle is a bit tricky if not all teams have been built and all start times assigned.

To compute the \(h\) variables, we count the number of days worked and we know that for any worker, the number of days worked + the number of days idle is always equal to \(nd\). Thus we can write the following constraints:

\[
\begin{align*}
\text{worked: } & \forall x, \text{worked}_{x} = \sum_{y} a_{x,y} w_{y}.d \\
\text{idle1: } & \forall x,d, h_{x,d} \Leftrightarrow g_{x,d} = 0 \\
\text{full schedule: } & \forall x, \text{worked}_{x} + \sum_{y} h_{x,y} = nd
\end{align*}
\]

With all the extra variables and constraints, we can now state the \textit{forbidden} constraint:

\[
\text{forbidden2: } \forall x,d \in [1..\#nd-2] \\
& f_{x,d} + h_{x,d+1} + f_{x,d+2} \leq 2
\]

This constraint, as opposed to the \textit{forbidden1} constraint, the implementation of \textit{forbidden2} is exact. It does not rule out valid solutions. Unfortunately, it propagates very late as only when the schedule for a worker finished is this constraint fired – because only at that time are the \(h\) variables completely defined.

Solving the Complete Problem

In this section, we investigate the effect of data size on the feasibility of the previous approaches and the different consumptions in term of memory and time.

Test Sets

To evaluate the different consumptions for the model, we have generated 4 tests sets of different size:

- **Tiny:** 20 workers and 60 jobs
- **Small:** 40 workers and 200 jobs
- **Medium:** 100 workers and 500 jobs
- **Large:** 800 workers and 2000 jobs. This is the size of the real world problem this model is inspired from.

All these test sets have 15 skills, 20 days. The \textit{far} predicate is implemented in the following way. All workers homes and all jobs locations are placed randomly on a 10×10 grid. Then we use a cutoff distance (6) and a manhattan distance.

Thus, one job \(y\) and one worker \(x\) ’s home are far from each other if and only if

\[
\text{abs}(j_{y}.\text{pos}X - w_{x}.\text{pos}X) + \text{abs}(j_{y}.\text{pos}Y - w_{x}.\text{pos}Y) > 6
\]

We will use these data sets to test ideas. As the large size is very challenging to solve, we cannot hope to test new ideas easily. Thus the need for smaller test sets to evaluate ideas before the polishing needed to solve the large instance.

Experimental Context

Due to various external constraints, the model has to be coded in ILOG OPL 5.2(OPL 2007) and the search part has to be very simple.

The goal here is find how we can solve a large and complex problem without writing complex search procedures or custom constraints.

Hitting the Size Limit

We evaluate our two implementations and the different test sets. For all experiments, we present the number of constraints in our engine used to solve it, the number of variables in the model, the memory used, the number of possible assignments – that is the number of pairs of compatible worker - job, and the number of possible interactions between two jobs, that is the number of times two jobs may share a worker. This will for instance count the number of disjunctions in the disjunctive model.

We begin with the disjunctive model on the tiny samples as any other size of sample will not fit into 1.5 GB memory. We tried with and without a simple shaving schema (as exposed in the next section).
We first report the disjunctive model on the tiny sample with and without shaving.

<table>
<thead>
<tr>
<th></th>
<th>no shaving</th>
<th>shaving</th>
</tr>
</thead>
<tbody>
<tr>
<td># constraints</td>
<td>2208</td>
<td>2045</td>
</tr>
<tr>
<td>memory (MB)</td>
<td>37</td>
<td>35</td>
</tr>
<tr>
<td># variables</td>
<td>1340</td>
<td>1340</td>
</tr>
<tr>
<td># assignments</td>
<td>488</td>
<td>293</td>
</tr>
<tr>
<td># interactions</td>
<td>3600</td>
<td>961</td>
</tr>
</tbody>
</table>

This implementation would not even create the model for other sizes of test sets (small, medium and large).

We move on to the agenda based implementation on the tiny test sets.

<table>
<thead>
<tr>
<th></th>
<th>no shaving</th>
<th>shaving</th>
</tr>
</thead>
<tbody>
<tr>
<td># constraints</td>
<td>10073</td>
<td>7192</td>
</tr>
<tr>
<td>memory (MB)</td>
<td>17.6</td>
<td>12</td>
</tr>
<tr>
<td># variables</td>
<td>2180</td>
<td>2180</td>
</tr>
<tr>
<td># assignments</td>
<td>3721</td>
<td>2857</td>
</tr>
<tr>
<td># interactions</td>
<td>40000</td>
<td>20736</td>
</tr>
</tbody>
</table>

And the on the agenda based implementation of the small test sets.

<table>
<thead>
<tr>
<th></th>
<th>no shaving</th>
<th>shaving</th>
</tr>
</thead>
<tbody>
<tr>
<td># constraints</td>
<td>62764</td>
<td>49805</td>
</tr>
<tr>
<td>memory (MB)</td>
<td>212</td>
<td>166</td>
</tr>
<tr>
<td># variables</td>
<td>10120</td>
<td>10120</td>
</tr>
<tr>
<td># assignments</td>
<td>3721</td>
<td>2857</td>
</tr>
<tr>
<td># interactions</td>
<td>40000</td>
<td>20736</td>
</tr>
</tbody>
</table>

and finally on the medium test sets.

<table>
<thead>
<tr>
<th></th>
<th>no shaving</th>
<th>shaving</th>
</tr>
</thead>
<tbody>
<tr>
<td># constraints</td>
<td>too large</td>
<td>too large</td>
</tr>
<tr>
<td>memory (MB)</td>
<td>too large</td>
<td>too large</td>
</tr>
<tr>
<td># variables</td>
<td>too large</td>
<td>too large</td>
</tr>
<tr>
<td># assignments</td>
<td>22792</td>
<td>20481</td>
</tr>
<tr>
<td># interactions</td>
<td>250000</td>
<td>190969</td>
</tr>
</tbody>
</table>

The large test set is not reachable with this model. For the medium test sets, only the shaving part is performed. The engine would not create the model and post constraints.

Discussion

The two model tested in this section performs very badly. We can analyse why.

On the disjunctive model, the problem comes from the implementation of the forbidden constraint. While the rest of the model is very light, this constraint set is not. In fact, if $p_1$ is the probability of a worker to be able to perform a job, then this worker may perform $\#j \times p_1$ jobs. If $p_2$ is the probability of a job to be far from home of a worker, then the number of forbidden constraints for a worker is ($\#j \times p_1 \times p_2)^2$.

Thus we have a total number of constraints in term of $\#j^2 \times \#w$. This is catastrophic.

If we look at the agenda based model, what is costly in the model is the agenda constraint itself. In the constraint, we have an element constraint:

$$\forall x,d, a_{x,y} \Rightarrow \bigvee_{\delta \in [0..j_y, d-1]} g_{x,t_y+\delta} = y$$

The $g_{s,t_y+d}$ part. This one is expensive because we have $\#w \times \#j \times \#d \times \text{average duration of these constraints}$. This means 160000 * 2 = 320000 constraints if the average duration of a job is 2. This is not as bad as before but still it will not even reach the medium instances (2,000,000 of this constraints).

Improving the Model

As we have seen before, solving the large model directly is not tractable. First we have improved the timetabling model and second we have investigated two possible ways of containing the complexity of the model.

There are different ways to reduce the size of the problem.

- The first one is exact and and is based on a real computation of feasible combination of workers to perform a job. With this information, we can rule out workers that never appear in any feasible combination.
- The second one is heuristic. We need a way to reduce the number of possibilities. We will implement two methods, one based on a limitation of the previous exact method and the second on a hybrid decomposition of the problem using a simplex to solve the assignment part.

Maintaining Active Jobs

The previous implementations of the scheduling were not satisfying. We worked on another one that would count active jobs on a given day. For this model, we reused the same $f$, $h$ and $g$ variables from the previous models:

$$\text{int } g_{x,d} \in [0..\#j]; x \in [1..\#w], d \in [1..\#d]$$

$$\text{bool } h_{x,d}; x \in [1..\#w], d \in [1..\#d]$$

$$\text{bool } f_{x,d}; x \in [1..\#w], d \in [1..\#d]$$

and we introduce a new kind of variables $e$ to decide if a job $y$ is active at a given date $d$.

$$\text{bool } e_{y,d}; y \in [1..\#j], d \in [1..\#d]$$

We can now post constraint that will maintain these $e$ variables:

$$\text{effective: } \forall y,d e_{y,d} = (d - y_y, d + 1 \leq t_y \leq d)$$

Which basically says that the time interval representing the job $y$ is spanning over the day $d$.

We can now implement the idle, far, valid schedule and forbidden constraints.

$$\text{valid schedule}_1: \forall x,d, \sum_y e_{y,d} \times a_{x,y} \leq 1$$

$$\text{idle}_2: \forall x,d, \sum_y e_{y,d} \times a_{x,y} + h_{x,d} = 1$$

$$\text{far}_2: \forall x,d, \sum_y e_{y,d} \times f(x,y) \times a_{x,y} = f_{x,d}$$

$$\text{forbidden}_3: \forall x,d \in [1..\#d-2], f_{x,d} + h_{x,d+1} + f_{x,d+2} \leq 2$$

This will force the corresponding $a_{x,y}$ variable to 0.
The valid schedule is a simple constraint. It states that at most one job is active for any given day and any given worker.

The idle is also simple as it states that a worker is either performing a job or idle. It is interesting to see that the $h$ variables are in fact the slack variables of the valid schedule constraints. In that case, the idle constraints subsumes the valid schedule constraint and the latter can be removed.

In the same spirit, the far constraint just checks if there is one far job active for a given worker and a given day.

The forbidden constraint is the same as the previous one.

This model is much better than the previous one in our case as the complexity depends on the number of time points, which is low in our case. Thus the discrete time approach is much lighter in memory than the disjunctive one.

Shaving Combinations of Workers

The scope of the skill covering constraint is limited to one assignment at a time. We have added another constraint that rules out workers that have no skills needed by the job:

\[
\text{exclusion: } \forall x, y, j. y \cap w_x, s = \emptyset \Rightarrow a_{x, y} = 0
\]

With this method, we can create a sub-model that will compute feasible solutions of the skill covering, card and exclusion constraints.

Now, we can embed this algorithm inside a script that will loop over feasible solutions and record workers selected by the sub-algorithm.

We can now experiment with this shaving module.

<table>
<thead>
<tr>
<th>solution limit</th>
<th>#possible</th>
<th># removed</th>
<th>run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>488</td>
<td>290</td>
<td>0.2</td>
</tr>
<tr>
<td>10</td>
<td>488</td>
<td>293</td>
<td>0.2</td>
</tr>
<tr>
<td>50</td>
<td>488</td>
<td>305</td>
<td>0.2</td>
</tr>
<tr>
<td>100</td>
<td>488</td>
<td>305</td>
<td>0.2</td>
</tr>
<tr>
<td>200</td>
<td>488</td>
<td>305</td>
<td>0.2</td>
</tr>
<tr>
<td>500</td>
<td>488</td>
<td>305</td>
<td>0.2</td>
</tr>
<tr>
<td>1000</td>
<td>488</td>
<td>305</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Then with the small test sets:

<table>
<thead>
<tr>
<th>solution limit</th>
<th>#possible</th>
<th># removed</th>
<th>run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3721</td>
<td>1153</td>
<td>1.0</td>
</tr>
<tr>
<td>10</td>
<td>3721</td>
<td>1265</td>
<td>1.2</td>
</tr>
<tr>
<td>50</td>
<td>3721</td>
<td>1454</td>
<td>2.0</td>
</tr>
<tr>
<td>100</td>
<td>3721</td>
<td>1485</td>
<td>2.4</td>
</tr>
<tr>
<td>200</td>
<td>3721</td>
<td>1502</td>
<td>2.7</td>
</tr>
<tr>
<td>500</td>
<td>3721</td>
<td>1513</td>
<td>2.8</td>
</tr>
<tr>
<td>1000</td>
<td>3721</td>
<td>1513</td>
<td>2.8</td>
</tr>
</tbody>
</table>

then with the medium test sets:

<table>
<thead>
<tr>
<th>solution limit</th>
<th>#possible</th>
<th># removed</th>
<th>run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>22792</td>
<td>4535</td>
<td>6.4</td>
</tr>
<tr>
<td>10</td>
<td>22792</td>
<td>5043</td>
<td>7.9</td>
</tr>
<tr>
<td>50</td>
<td>22792</td>
<td>6241</td>
<td>18.7</td>
</tr>
<tr>
<td>100</td>
<td>22792</td>
<td>6301</td>
<td>28.2</td>
</tr>
<tr>
<td>200</td>
<td>22792</td>
<td>6313</td>
<td>39.9</td>
</tr>
<tr>
<td>500</td>
<td>22792</td>
<td>6325</td>
<td>53.7</td>
</tr>
<tr>
<td>1000</td>
<td>22792</td>
<td>6334</td>
<td>57.3</td>
</tr>
</tbody>
</table>

and finally with the large test sets:

<table>
<thead>
<tr>
<th>solution limit</th>
<th>#possible</th>
<th># removed</th>
<th>run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>762596</td>
<td>85734</td>
<td>258</td>
</tr>
<tr>
<td>10</td>
<td>762596</td>
<td>87225</td>
<td>320</td>
</tr>
<tr>
<td>50</td>
<td>762596</td>
<td>140997</td>
<td>793</td>
</tr>
<tr>
<td>100</td>
<td>762596</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>200</td>
<td>762596</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>500</td>
<td>762596</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>1000</td>
<td>762596</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

The idea to limit the loop is useful in practice and allow a correct shaving and a robust one in term of runtime if we restrict ourselves to small limits (less than 50).

Furthermore, the sheer numbers displayed illustrates the complexity of the problems. In the large instances, 762596 possible assignments is simply to big.

Limit Combinations of Workers

The idea is to change the behavior of the shaving procedure when the solution limit is crossed. In that case, instead of recording the possible assignments, we record the assignments found in the previous solution.

Thus we limit the possible combination and remove feasible solutions from the model. On the other hand, we will get a much smaller problem. In that sense, it is interesting to look at small values for the loop limit.

First with the tiny test sets
The good effect is that it simplifies a lot the scheduling model. We will evaluate these techniques in the results section.

**Hybrid Implementation**

The idea here is to use ILOG CPLEX to solve the packing + set covering problem. More specifically the *card, day worked, skill covering* and *exclusion* constraints. The following unique assignment is then given to the schedule.

This method can be seen as an optimized shaving version. The good effect is that it simplifies a lot the scheduling module.

Here are the remaining constraints (we note $\beta_{x,y}$ if the assignment (worker $x$ on job $y$) is selected by the planning. This is now a data and not a variable anymore):

<table>
<thead>
<tr>
<th>solution limit</th>
<th>#possible</th>
<th># removed</th>
<th>run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>488</td>
<td>366</td>
<td>0.2</td>
</tr>
<tr>
<td>6</td>
<td>488</td>
<td>343</td>
<td>0.2</td>
</tr>
<tr>
<td>10</td>
<td>488</td>
<td>322</td>
<td>0.2</td>
</tr>
<tr>
<td>30</td>
<td>488</td>
<td>305</td>
<td>0.2</td>
</tr>
<tr>
<td>60</td>
<td>488</td>
<td>305</td>
<td>0.2</td>
</tr>
<tr>
<td>100</td>
<td>488</td>
<td>305</td>
<td>0.2</td>
</tr>
<tr>
<td>300</td>
<td>488</td>
<td>305</td>
<td>0.2</td>
</tr>
</tbody>
</table>

This technique shows good results in reducing the total size of the model. We will evaluate these techniques in the results section.

**Experimental Results**

It is time now to evaluate these two new models on the different test sets.

All experiments are made with ILOG OPL 5.2. They are made on an Intel quad 2.67 GHz Xeon with 4 GB of memory running Fedora 7 (64 bit).

**Results with Limited Combinations of Workers**

We give the results with the full model and the model limited with a solution limit of six and three. The time limit is 2s per job, thus 120, 400, 1000 and 4000s.

Here is the tiny test set with a solution limit of 3 and 6:

<table>
<thead>
<tr>
<th>Tiny test set</th>
<th>Full</th>
<th>Limited 6</th>
<th>Limited 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory (MB)</td>
<td>7.1</td>
<td>6.4</td>
<td>6.2</td>
</tr>
<tr>
<td>Best Solution found</td>
<td>48</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>Time (s)</td>
<td>11</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Planning Solution</td>
<td>48</td>
<td>48</td>
<td>48</td>
</tr>
</tbody>
</table>

and for the small test set:

<table>
<thead>
<tr>
<th>Small test set</th>
<th>Full</th>
<th>Limited 6</th>
<th>Limited 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory (MB)</td>
<td>52</td>
<td>32</td>
<td>28.8</td>
</tr>
<tr>
<td>Best Solution found</td>
<td>111</td>
<td>142</td>
<td>145</td>
</tr>
<tr>
<td>Time (s)</td>
<td>350</td>
<td>347</td>
<td>348</td>
</tr>
<tr>
<td>Planning Solution</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
</tbody>
</table>
Thus it is the complexity of the problem that forbids the a good search strategy. The search is lost and the number of constraints is so huge that we just do not search enough. We tried more aggressive search strategies, ones that would try to perform all jobs instead of one that would first try not to perform any job and then perform more and more of them (branch up instead of branch down on the \(b_y\) variables). But this one is not robust enough and while very good solutions are found on the small and tiny test sets, we do not find any solution for the medium and large instances.

Finally, the hybrid solution is by far the best and most robust approach. It consumes less memory, finds good solution. Still, on the large instances, there is room for improvement as we are quite far (1759 vs 3327).

### Conclusion

The story repeats itself. We tried to get rid of the distinction between planning and scheduling on this timetabling problem and we failed.

The combinatorial explosion of the search space and of the number of constraints are the main limiting factors. As a result, the problem cannot be solved by one engine in a single run. Decomposition have to be used.

Furthermore, we are a bit disappointed by the results of the model with limited combination of workers. This is particularly visible on the medium data set where the obtained results (around 150) are very far from the planning solution (510).

If we look at the bright side, the hybrid solution is very small and elegant. It finds optimal solutions quickly for all instances except the large ones. And for the large instances, it finds good solutions and we are confident we will find a way to solve the problem effectively with a little bit of tweaking.

Finally, in order to sparkle discussion and comparison with other methods, we have decided to make the instances public. They can be obtained upon request from the author. Please note that we are working on a more complex version of the problem where some days are unavailable for workers.

This will be the subject of future work.

### References


Drexl, A.; Frahm, J.; and Salewski, F. Audit-staff scheduling by column generation.


**Acknowledgements**

I would like to thank Alex Fleisher, Frank Wagner, Philippe Refalo, Olivier Lhomme and Frédéric Delhoume for their contribution to this work.

**Annex**

Here is the tuple definition in ILOG OPL 5.2(OPL 2007)

```cpp
tuple Assignment {
  int duration;
  int required;
  int weight;
  int posX;
  int posY;
  int skills[allSkills];
}

tuple Worker {
  int homeX;
  int homeY;
  int qualifications[allSkills];
}
```

and here is what a data test looks like

```cpp
nbWorkers = 20;
nbJobs = 60;
nbSkills = 15;
nbDays = 20;
assignments = [
  <3 4 1 2 1 [1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1] >
  ...
];
workers = [
  <7 3 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] >
  ...
];
```
Feasible Distributed CSP Models for Scheduling Problems*

Miguel A. Salido
Department of Information Systems and Computation
Technical University of Valencia, Spain
msalido@dsic.upv.es

Abstract
Nowadays, many real problems can be formalized as Distributed CSPs. A distributed constraint satisfaction problem (DisCSP) is a CSP in which variables and constraints are distributed among multiple automated agents. Many researchers assume for simplicity that each agent has exactly one variable. For real planning and scheduling problems, these distributed techniques require a large amount of messages passed among agents, so these problems are very difficult to solve. In this paper, we present a general distributed model for solving real-life scheduling problems and propose some guidelines for distributing large-scale problems. Furthermore, we present two case studies in which two scheduling problems are distributed by using our model.

Introduction
In recent years we have seen an increasing interest in Distributed Constraint Satisfaction Problem (DisCSP) formulations to model combinatorial problems (see the special issue on Distributed Constraint Satisfaction in the Artificial Intelligence journal, vol 161, 2005). There is a rich set of real-world distributed applications, such as network systems, planning, scheduling, resource allocation, etc, for which the DisCSP paradigm is particularly useful. In such distributed applications, privacy issues, knowledge transfer costs, robustness against failure, etc preclude the adoption of a centralized approach (Faltings & Yokoo 2005).

Briefly, a CSP consists of: a set of variables $X = \{x_1, x_2, ..., x_n\}$; each variable $x_i \in X$ has a set $D_i$ of possible values (its domain); a finite collection of constraints $C = \{c_1, c_2, ..., c_P\}$ restricts the values that the variables can simultaneously take.

A solution to a CSP is an assignment of values to all the variables so that all constraints are satisfied; a problem with a solution is termed satisfiable or consistent.

A distributed CSP is a CSP in which the variables and constraints are distributed among automated agents (Yokoo & Hirayama 2000). Finding a value assignment to variables that satisfies inter-agent constraints can be viewed as achieving coherence or consistency among agents.

The most cited papers related to DisCSP make the following assumptions for simplicity in describing the algorithms:
1. Each agent has exactly one variable.
2. All constraints are binary.
3. Each agent knows all constraint predicates relevant to its variable.

Although the great majority of real problems are naturally modelled as non-binary CSPs, the second assumption is comprehensible due to the fact that there exist some techniques that translate any non-binary CSP into an equivalent binary one (Bacchus & van Beek 1998).

However, the first assumption is too restrictive and the main basic research focuses on small instances. Also, little work has been done to solve real-life problems.

Main Features in DisCSPs
If all knowledge about the problem could be gathered into one agent, this agent could solve the problem alone using traditional centralized constraint satisfaction algorithms. However, such a centralized solution is often inadequate or even impossible. Faltings and Yokoo (Faltings & Yokoo 2005) present some reasons why distributed methods may be desirable:

- The cost of creating a central authority. A CSP may be naturally distributed among a set of agents. In such cases, a central authority for solving the problem would require adding an additional element that was not present in the architecture. Examples of such systems are sensor networks or meeting scheduling.
- The knowledge transfer costs: In many cases, constraints arise from complex decision processes that are internal to an agent and cannot be articulated to a central authority. Examples of this range from simple meeting scheduling, where each participant has complex preferences that are hard to articulate, to coordination decisions in virtual enterprises that result from complex internal planning. A
centralized solver would require such constraints to be completely articulated for all possible situations. This would entail prohibitive costs.

- Privacy/Security concerns: Agents involve constraints that may represent strategic information that should not be revealed to competitors, or even to a central authority. This situation often arises in many enterprises. Privacy is easier to maintain in distributed solvers.

- Robustness against failure: The failure of the centralized server can be fatal. In a distributed method, a failure of one agent can be less critical and other agents might be able to find a solution without the failed agent. Such concerns arise, for example, in sensor networks, but also in web-based applications where participants may leave while a constraint solving process is ongoing.

These reasons have motivated significant research activity in distributed constraint satisfaction. Up to now, the field has reached a certain maturity and has developed a range of different techniques. Nevertheless, most of the works are focused on developing new techniques which are evaluated using toy problems and random benchmarks.

**Open Issues in DisCSPs**

In spite of significant progress, there are many important open issues in distributed CSP. The five main open issues for using distributed CSP are the followings:

- While distributed algorithms eliminate the need for a central authority, the currently known algorithms pay a high price in efficiency. In general, the message traffic even for a single agent can be higher than what would be required to communicate the entire problem to a leader agent that could solve it centrally. More research is required to significantly reduce the communication requirements, possibly with radically different algorithms that are better suited for distribution.

- Many DisCSP algorithms assumes an agent has enough knowledge to evaluate constraints that are related to its variables. If this is not true, some constraints may still have to be communicated or additional communication may be needed. Also, more research is needed on algorithms that minimize the number of constraint evaluations when evaluating constraints is costly.

- While there are algorithms using cryptographic techniques to ensure complete privacy of agent constraints, their message complexity is very high. For most other DisCSP algorithms, there is no good characterization of how much information is revealed to other agents. More research is needed on measures of privacy loss and on algorithms that balance the trade-off between privacy loss and efficiency.

- While most distributed algorithms tolerate certain kinds of agent failures, there is no good characterization of the kind of failures that are allowed for each algorithm. In general, this issue has not yet been given significant attention in research.

- While most distributed algorithms manage only one variable per agent, there are no specific distributed algorithms to solve real-world problems in an efficient way. This issue has not yet been given enough attention in research. This paper focuses on this issue.

Other issues that are important but have received little attention so far include openness, i.e., the possibility to add and remove agents dynamically during execution, and incentive-compatibility, i.e., making algorithms safe against manipulation by self interested agents.

**From Basic Research Toward Applied Research**

One of the pioneer researchers in DisCSP said "So far, we assume that each agent has only one local variable. Although the developed algorithms can be applied to the situation where one agent has multiple local variables by the following methods, both methods are neither efficient nor scalable to large problems" (Yokoo & Hirayama 2000).

- Method 1: each agent finds all solutions to its local problem first. By finding all solutions, the given problem can be re-formalized as a distributed CSP, in which each agent has one local variable whose domain is a set of obtained local solutions. Then, agents can apply algorithms for the case of a single local variable. The drawback of this method is that when a local problem becomes large and complex, finding all the solutions of a local problem becomes virtually impossible.

- Method 2: an agent creates multiple virtual agents, each of which corresponds to one local variable, and simulates the activities of these virtual agents. However, since communicating with other agents is usually more expensive than performing local computations, it is wasteful to simulate the activities of multiple virtual agents without distinguishing the communications between virtual agents within a single real agent, and the communications between real agents.

In spite of significant progress in distributed CSP, the following question is straightforward: Why is only one variable per agent assumed? (Salido 2007). Only some works include a set of variables into an agent (Silaghi & Faltings 2005),(Ezzahir & Bouyakhf 2007),(Salido & Barber 2006). Nevertheless, few works have been focused on distributed techniques for solving large scale problems (Yokoo et al. 1998). In this paper, we present different alternatives for managing large scale problems. Each agent will be committed to a large number of variables and constraints and several subproblems can be executed concurrently depending on the internal structure.

**A General Distributed Model**

Depending on the problem to be considered, the distributed model will maintain different properties. Our general model for solving scheduling problems can be considered as a synchronous model. It is meant to be a framework for interacting holons/agents to achieve a consistent state. The main idea of our holonic/multi-agent system model is based on
(Salido, Giret, & Barber 2003) in which the problem is partitioned into a set of subproblems; then the subproblems are classified in the appropriate order and are solved concurrently. In this section, we introduce the notion of holon as a complementary idea of agent. Depending on the scheduling problem, we will use holons instead of agents. A holon is an autonomous and cooperative unit that can be seen as a whole and a part (Koestler 1971). Therefore, a holarchy is a group of basic holons and/or recursive holons that are themselves holarchies. A Holonic architecture (HMS 1994) (Koestler 1971) is committed to organizing entities (holon or agent (Giret & Botti 2004)) that are responsible for solving each subproblem.

![Figure 1: General Distributed Model.](image)

Depending on the problem, it is partitioned in $k$ blocks or clusters in order to be studied by holons/agents called block agents. Furthermore, a partition agent is committed to classifying the subproblems in the appropriate order depending on the selected proposal.

Once the problem is divided into $k$ blocks by a preprocessing agent, a group of block agents concurrently manages each block of constraints. Each block agent/holon is in charge of solving its own subproblem by means of a search algorithm. Each block agent/holon is free to select any algorithm to find a consistent partial state. It can select a local search algorithm, a backtracking-based algorithm, or any other, depending on the problem topology. In any case, each block agent/holon is committed to finding a solution to its particular subproblem. This subproblem is composed by its CSP subject to the variable assignment generated by the previous block agents/holons. Thus, block agent/holon 1 works on its group of constraints. If block agent/holon 1 finds a solution to its subproblem, then it sends the consistent partial state to block agent/holon 2, and they both work concurrently to solve their specific subproblems; block agent/holon 1 tries to find another solution and block agent/holon 2 tries to solve its subproblem knowing that its common variables have been assigned by block agent/holon 1. This second solution found by block agent/holon 1 is stored to be sent to block agent/holon 2 if it was necessary. In this case, block agent/holon 2 will not wait for block agent/holon 1 to search for a new solution.

Thus, block agent $j$, with the variable assignments generated by the previous block agents/holons, works concurrently with the previous block agents/holons and tries to find a more complete consistent state using a search algorithm. Finally, the last block agent/holon $k$, working concurrently with block agents/holons 1, 2, ..., $(k-1)$, tries to find a consistent state in order to find a problem solution.

Figure 1 shows the holonic system, in which the preprocessing agent carries out the network partition and the block agents/holons (ai) are committed to concurrently finding partial problem solutions (si,j, where $i$ denotes the number of block agents/holons and $j$ the jth solution). Each block agent/holon sends the partial problem solutions to the following block agent/holon until a problem solution is found (by the last block agent/holon). For example, state: $s_{11} + s_{21} + ... + s_{k1}$ is a problem solution. The concurrence can be seen in Figure 1 in Time step 6 in which all block agents/holons are concurrently working. Each block agent/holon maintains the corresponding domains for its new variables. The new variables are the variables that are not involved in previous block agent/holon. The block agent/holon must assign values to its new variables so that the block of constraints is satisfied. When a block agent/holon finds a value for each new variable, it then sends the consistent partial state to the next block agent/holon. When the last block agent/holon assigns values to its new variables satisfying its block of constraints, then a solution is found. It must be taken into account that if a block agent/holon maintains too many variables or constraints, it can be decomposed into a set of new agents/holons.

### Some Guidelines for Distributing Large-Scale Problems

To solve large-scale problems, we must distribute the problems taking into account some guidelines:

1. **The number of subproblems (agents).** They must be in concordance with the size of the problem. As we have pointed out in the previous sections, one agent per variable is unmanageable. A problem with thousands of variables and constraints cannot be modelled as a distributed model with thousands of agents due to the high computational cost in the solving process. Generally and due to privacy issues, the number of subproblems is straightforward, given by the nature of the problems. Nevertheless, some subproblems are too large and they can be divided/decomposed again into smaller ones in order to be solved in a reasonable time. A reasonable way to divide the problem is by means of graph partitioning techniques (Salido & Barber 2006). However, in many real problems, the best way to partition the problem is by carrying out a domain dependent partition. Following, we present an example of railway scheduling problem distributed by types of trains and sets of stations.

2. **The order in which each subproblem is executed (agent
priority). Sometimes all subproblems can be executed concurrently and partial states are sent to their neighbours. In many other cases, the subproblems can be ordered using a selected criteria.

- From the subproblem with the most neighbours to the subproblems with the least neighbours.
- From the tightest subproblems to the loosest ones.
- From the subproblems that maintains hard constraints to the subproblems with soft constraints.
- etc...

This ordering does not mean that the subproblems are solved sequentially, but rather that some subproblems are executed first and then all neighbours are concurrently executed (see Figure 1). Furthermore, this ordering, represented by priorities among agents, can change dynamically depending on many factors such as number of no-goods, number of backtracks, etc.

3. The management of backtracking. When an agent does not find a partial solution, it must communicate its current state to the related agents in order to avoid unnecessary searches. Some works decompose the subproblem into a set of subproblems in such a way that the resultant problem is represented as a tree. Each node in the tree is a subproblem (Abril, Salido, & Barber 2007). Once a node/agent finds a partial solution, it is sent to its children, and they are solved concurrently. However, if a node/agent does not find a partial solution, a message is sent to its parents, and, depending on the management of the backtracking (which parent it backtracks), the search space will be pruned more efficiently.

4. The necessity of a central authority. As we have pointed out, depending on the problem, a central regulatory authority will be necessary or not. Many researchers suggest that for some multi-agent systems, no central regulatory authority is needed and can be replaced by a virtual representation where each agent is responsible for maintaining its own partial view of the relevant institutional state. This is due to the fact that, if the central authority fails, the problem cannot be solved. However, many real problems are hierarchical by nature and it is necessary to generate different levels of abstraction. In the next section, we will present a global road transportation system: a hierarchical system that maintains as many levels as necessary, depending on the problem complexity.

Distributed Models for Distributed Scheduling Problems: Two case studies

Many real problems are distributed by nature. However, this distribution does not imply one variable per agent. For instance, many scheduling problems are decentralized by nature, and the problem is decomposed in clusters. Each cluster (composed by variables and constraints) is solved by an agent, and it communicates a consistent partial state. In this way, a large-scale scheduling problem can be solved with reasonable efficiency, maintaining all privacy issues. Most of these problems are domain-dependent and general distributed models (as the model proposed above) are not appropriate. Therefore, domain dependent distributed models must be developed to efficiently manage these problems.

In this section, we present two real-life scheduling problems, which are very complex problems and are distributed by nature. To solve them in a distributed way, we group several compatible variables per agent so that the problem can be solved in a reasonable time.

Railway Scheduling Problem

Train timetabling is a difficult problem, particularly in the case of real networks, where the number of constraints and the complexity of constraints grow drastically. A feasible train timetable should specify the departure and arrival time of each train to each location of its journey in such a way that the line capacity and other operational constraints are taken into account. Traditionally, train timetables are generated manually by drawing trains on the time-distance graph. The train schedule is generated from a given starting time and is manually adjusted so that all constraints are met. High priority trains are usually placed first followed by lower priority trains. It can take many days to develop train timetables for a line, and the process usually stops once a feasible timetable has been found. The resulting plan of this procedure may be far from optimal.

The literature of the 1960s, 1970s, and 1980s related to rail optimization was relatively limited. Compared to the airline and bus industries, optimization was generally overlooked in favor of simulation or heuristic-based methods. However, Cordeau et al. (Cordeau, Toth, & Vigo 1998) point out greater competition, privatization, deregulation, and increasing computer speed as reasons for the more prevalent use of optimization techniques in the railway industry. Our review of the methods and models that have been published indicates that the majority of authors use models that are based on the Periodic Event Scheduling Problem (PESP) introduced by Serafini and Ukovich (Serafini & Ukovich 1989). The PESP considers the problem of scheduling as a set of periodically recurring events under periodic time-window constraints. The model generates disjunctive constraints that may cause the exponential growth of the computational complexity of the problem depending on its size. Schrijver and Steenbeek (Schrijver & Steenbeek 1994) have developed CADANS, a constraint programming- based algorithm to find a feasible timetable for a set of PESP constraints. The train scheduling problem can also be modeled as a special case of the job-shop scheduling problem (Silva de Oliveira (Silva de Oliveira 2001), Walker et al. (Walker & Ryan 2005)), where train trips are considered as jobs that are scheduled on tracks that are regarded as resources.

We modelled the railway scheduling problem as a Constraint Satisfaction Problem (CSP) and it was solved by using constraint programming techniques. However, due to the distributed nature of the problem and the huge number of variables and constraints that this problem generates, a distributed model was developed to distribute the resultant CSP into semi-independent subproblems so that the solution can be found efficiently (Salido et al. 2007).
Here, we present two ways for distributing the railway scheduling problem. It is partitioned into a set of subproblems by means of types of trains and by means of contiguous constraints.

**Distribution by types of trains** This distributed model is based on dividing the original railway problem by means of train types. In this model, each agent is committed to assigning values to variables related to a train or sets of trains in order to minimize the journey time. This partition model takes into account some of the guidelines given above.

- Depending on the problem instance, the number of partitions will be given by the railway operator or by the number of trains. Figure 2 shows a running map with 20 partitions. Each agent manages one train. Each train generates a large number of variables and constraints, depending on the number of stations and the user requirements. Furthermore, this model allows us to improve privacy. Currently, due to the policy of deregulation in the European railways, trains from different operators work on the same railway infrastructure. In this way, the partition model also gives us the possibility of partition the problem so that each agent is committed to an operator. Thus, different operators maintain privacy about strategic data.

- This model allows us to efficiently manage priorities between different types of trains (regional trains, high-speed trains, freight trains). In this way, agents committed to priority trains (high-speed trains) will first carry out value assignment to variables in order to achieve better journey times. This ordering is inserted into our distributed model to solve the scheduling problem concurrently.

**Distribution by set of stations** This distributed model is based on distributing the original railway problem by means of contiguous stations. The deregulation of European railway operators gives the opportunity to schedule long journeys. However, long journeys involve large number of stations in different countries with different railway policies. Therefore, a logical partition of the railway network can be carried out by means of regions (contiguous stations).

Some of the guidelines presented above that must be taken into account in this model are:

- The number of subproblems depends directly on several factors: distance of the journey, number of different regions (mainly countries) and railway topology. This distributed model divides the problem into a set of physical regions. It is important to analyze the railway infrastructure in order to detect restricted regions (bottlenecks). To balance the problem, each agent is committed to a different number of stations. An agent can manage many stations if they are not restricted stations; however an agent can manage only a few stations if they represent bottlenecks.

- The order in which each subproblem is executed plays an important role. Agents committed with bottlenecks have preference to assign values to variables due to the fact that their domains are reduced (variable ordering). Once the agent with the tightest constraints solves its subproblems, both the previous agent and the following agent must concurrently solve their own subproblems.

- The management of backtracking is very important to avoid unnecessary constraint checking. For instance, if agent 4 in Figure 3 finds a partial solution to its subproblem, it communicates to agent 3 and agent 5. Both agents concurrently search to find their own partial solutions. However, if agent 3 does not find a partial solution, it sends a nogood message to agent 4, and this last agent sends a message to agent 5 in order to stop its search. As we pointed out in Figure 1, while agent 3 and agent 5 work concurrently to find their partial solutions, agent 4 also works in the same time step in order to find another partial solution. This last partial solution will be used if agent 3 or agent 5 backtrack due to inconsistency with the previous partial solution given by agent 4.

Figure 3 shows the journeys to be scheduled between two cities. They are decomposed into several shorter journeys. The set of stations are partitioned in blocks of contiguous
stations and a set of agents will coordinate with each other to achieve a global solution. Thus, we can obtain important results such as railway capacity (Abril et al. 2007), consistent timetable, etc.

**Global Road Transportation System**

During recent years the development of automated traffic systems has received increased attention, and substantial effort has been invested in trying to find a solution to problems associated with road transport. Among these problems are road accidents caused by human-related factors, such as tiredness, loss of control, a slow reaction time, limited field of view, etc. A further transport-related problem is that of loss of time which may be caused by slow driving speed due to weather conditions, road conditions, visibility, and traffic congestion. In this section, we present a global road transportation system, which is being developed by several European Universities. The main goal of the algorithmic section is to develop algorithms capable of creating driving schemes for a vehicle from any arbitrary address to any other arbitrary address (in the address space of the system), while considering, and if necessary adapting, the driving schemes of other vehicles travelling in the system at the same time according to priorities, driving and optimization rules. Thus, distributed techniques are necessary for solving these problems.

The Global Automated Transportation System (GATS) (Zelinkovskyn) is a driverless, integrated transport system. It has the astonishing ability to simultaneously coordinate the macro and micro needs of road transport networks. Millions of vehicles can be optimally, simultaneously and automatically "driven" over a virtually unlimited geographic region, including whole continents, while the requirements of each individual vehicle and its passengers are attended to at the same time. It is an innovative concept, based on simple, recognized principles and proven technologies. Its application will revolutionize road travel by dramatically increasing safety, reducing congestion, and eliminating driving-associated stress and fatigue. The consequence...an overall improvement in the quality of everyday life.

Due to the decentralized and modular nature of the architecture it can be implemented with the same ease and simplicity in both small contained areas such as airports and theme parks and in larger areas such as local, national and international road systems.

![Figure 4: Driving on the System.](image)

Following, we summarize the architecture above and below the road. In the center of a traffic lane, 15-20 cm below the road surface there is an "Intelligent Cable" of about 1 cm in diameter which is comprised of tiny intelligent transponders (Road-Units (RUs)) located at fixed distances (less than a vehicle length of 3m) from each other. While driving, the vehicle sends short radio transmissions down towards the RUs at regular time intervals (about every 30 milliseconds). The RU receives a transmission, processes it, and responds with a radio transmission back to the vehicle. The vehicle communicates continuously with the RUs one after the other incessantly. Thus, it has continuous radio communication with the RUs and the whole system connected to them. The interchange of radio transmissions between the vehicles and the RUs also facilitates lateral and longitudinal positioning of vehicles on the road, as presented in Figure 4.

The memory of each RU stores the specifications of the RU and individual driving instructions that it will transmit to each vehicle above it. Several hundreds of consecutive RUs constitute a Segment, whose functions are administered by a Segment Controller. The Segment Controller is connected to its RUs through the Parallel Buses and is responsible for "driving" the vehicles passing in its domain; performing routine maintenance check-ups of the components in its segment; and monitoring and regulating their mutual performance. A group of adjacent Segment Controllers has a superior controller, the Level-1 Controller, which coordinates and controls its individual and mutual functions. A group of adjacent Level-1 Controllers has a superior controller: Level-2 Controller. This goes on hierarchically (Figure 5). The Top Level Controller coordinates and controls the functions of the whole system. There is virtually no limit to the number of levels and to the size of the geographic domain of the systems.

![Figure 5: Hierarchical Architecture of the System.](image)

**Integrated Functioning** Assume a vehicle is in a parking lot above a RU. The passengers turn the vehicle on, which begins to send short radio transmissions down towards the road. The RU detects those transmissions and responds with radio transmissions back to the vehicle. The RU initiates a communication session with the Segment Controller in order to inform it about the new event. The passengers in the vehicle enter their requirements as destination, priority, preferred routes etc. The vehicle’s processor sends a message to the Segment Controller which includes the requirements, the exact location of the vehicle relative to the RU and its own specifications. The Segment Controller processes the
request while considering additional inputs from other RUs in the Segment and from its superior Controller. Finally it prepares a driving instruction message for each RU in the Segment. The RUs will send these instructions to the vehicle when it passes above them. Each message includes an addressee (RU1 etc.), a vehicle ID, the expected arrival time of the vehicle to the RU, the speed that the vehicle should travel at and the driving direction. When the vehicle is driving from one RU to another, the active RU uses the Serial Bus to inform the next and previous RUs in the sequence about the exact timing, the ID number and other specifications of the moving vehicle. If the RUs detect intolerable deviation from the plan, they can initiate a so-called Emergency Braking Procedure. The active RU uses the Parallel Bus to inform the Segment Controller with information about the moving vehicle.

A Distributed Model for GATS Traditional Centralized techniques fail to model and implement problems of this type due to their complex and large nature. Due to the decentralized and modular nature of the architecture, the algorithms to obtain the scheduling of each vehicle must be distributed. Figure 6 shows the map of Europe to be distributed/divided into regions (countries); each region is divided into sub-regions, and so on.

![Figure 6: Map of Europe to be distributed.](image)

Briefly, the system is composed by a network, where nodes are locations and arcs are roads. Depending on the granularity, nodes are points in the road or regions in a country. In the lower level of the system, each RU is represented by a variable (see Figure 7). The system may be composed of millions of RUs. As we have explained in the first section, this problem cannot be managed by current distributed CP techniques by using a variable per agent. By using these approaches, the problem generates millions of agents and messages passing in the interaction scenarios. This makes the resulting DisCSP unmanageable.

To overcome these weakness, we use our distributed model in which the problem is partitioned into subproblems that represent regions, countries, etc (see Figure 7). Here, we use a Holonic architecture (HMS 1994)(Koestler 1971) to organize the entities (holon or agent (Giret & Botti 2004)) that are responsible for solving each subproblem. As we have pointed out above, a holon is an autonomous and cooperative unit that can be seen as a whole and a part (Koestler 1971).

The distributed model generated for this scheduling problem follows the guidelines presented in the paper.

- The number of subproblems depends on the size of the system. A holon can represent a track between two traffic lights or represent a region or a country. Figure 7 shows two holons that represent two countries, Spain and Italy. Each of them is be composed of a set of sub-holons that represent regions, and each sub-holon is composed by new sub-holons that represent sub-regions and so on. The base case is composed of individual variables that represent RUs.

- The execution of the subproblems is carried out in two steps. First, given the requirements of the passenger, (the destination is the most important requirement), the central authority is the Level i Controller that involves both origin and destination. This Level i Controller is committed to solving the shortest path in a high level problem (each node is a region). This path is only a first approach that guides us to find the real shortest path. Thus, Level i Controllers is executed first, then all Level i-1 Controller are executed concurrently and so on. Depending on the size of the journey, several hierarchical levels are necessary. Finally, the calculated route is sent to the Segment Controllers that are involved.

- Due to the dynamic structure of the problem, some parts of the system may change and new schedules must be calculated. The rescheduling is only calculated from the incidence to the destination. The management of backtrack-

![Figure 7: Distributed model with a central authority.](image)
ing is carried out in a way similar to the railway scheduling problem distributed by stations.

- As we have pointed out, the nature of the system makes the presence of a central authority necessary. However, due to the scalability of the system, the central authority has the same behaviour as a level controller. The central authority is the minimal level controller that involves both origin and destination. This level depends on the problem instance.

Conclusions

In the paper, we question the common assumption made in DisCSP literature in which each agent has just a single variable. Many real problems can be modelled as a CSP, but they cannot be solved by using DisCSP techniques due to the exploitation in message passing. Thus, new distributed techniques must be developed to solve large instances of real problems. In this paper, we present a general distributed model for solving large-scale problems and some guidelines for distributing these problems by relaxing the above assumption. We present two real-life problems which can be modelled as a distributed problem. They manage several variables per agent in order to solve these problems in a reasonable time. These problems follow some of the presented guidelines.

References


A Distributed CSP Approach for Solving Multi-agent Planning Problems

Oscar Sapena, Eva Onaindía, Antonio Garrido, Marlene Arangu
Dpto. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Valencia, Spain
osapena,onaindia,agarridot,marangu@dsic.upv.es

Abstract

Distributed or multi-agent planning extends classical AI planning to domains where several agents can plan and act together. There exist many recent developments in this discipline that range over different approaches for distributed planning algorithms, distributed plan execution processes or communication protocols among agents. One of the key issues about distributed planning is that it is the most appropriate way to tackle certain kind of planning problems, specially those where a centralized solving is infeasible. In this paper we present a new planning framework aimed at solving planning problems in inherently distributed domains where agents have a collection of private data which cannot share with other agents. However, collaboration is required since agents are unable to accomplish its own tasks alone or, at least, can accomplish its tasks better when working with others. Our proposal motivates a new planning scheme based on a distributed search of heuristic information and on a constraint programming resolution process.

Introduction

Distributed planning is the problem of finding a course of actions that will help a set of agents collectively satisfy certain desired goals. Due to an inherent distribution of resources such as knowledge and capability among the agents, an agent in a distributed planning system is unable to accomplish its own tasks alone, or at least can accomplish its tasks better when working with others (Durfee 2001). Distributed planning is still an open challenge, and there is an increasingly number of applications that can benefit from this research area: cooperative robotics (Wehowsky, Block, & Williams 2005) (Sirin et al. 2004), composition of semantic web services (Wu et al. 2003), manufacturing systems (Hahndel, Fuchs, & Levi 1996), etc.

The literature cites many reasons for which multi-agent planning is an interesting approach to pursue. One of these reasons is to split the problem into smaller subproblems which are usually easier to solve. This divide-and-conquer approach has been used in several distributed planning proposals (Rehak, Pechoucek, & Volf 2006) (Cox, Durfee, & Bartold 2005). In these approaches, multiple agents plan to achieve their individual goals independently and, then, these individual plans are merged into a global plan. The coordination/merging process is usually the most costly part since it is necessary to avoid cross-working or duplicating effort.

Another reason that often comes up is that of privacy (van der Krogt 2007). Especially in circumstances where the agents represent companies, sharing data with other parties is considered undesirable. At the same time, it is well recognized that cooperation may be mutually beneficial to all parties. In this paper we present a new planning framework aimed at solving problems of this type. Specifically, we will address problems with the following characteristics:

- **Distributed domains.** In these domains there exists an inherent distribution of resources such as knowledge and capability among the agents. This way, agents are clearly identified, so a problem decomposition stage is not required.

- **Privacy.** Agents maintain a set of private data, which are the beliefs that the agent will never share with other agents. The goals of an agent are also private, although it may require help from other agents to achieve them.

- **Collaboration.** In this framework, an agent often needs help from other agents to achieve the necessary conditions for executing an action. However, even though actions are jointly planned, they are individually executed, that is, agents do not get synchronized to carry out a same action jointly (for example, several robots pushing a single block together into a target area). This latest type of coordination is usually addressed in team-oriented planning approaches, where several agents collaborate to achieve a common global goal.

The presented approach is useful to solve planning problems in inherently distributed domains where a centralized solving process is not affordable. Additionally, agents have a collection of private data which cannot share with other agents so information exchange among them can only be achieved through the public or sharable data. Our proposal motivates a new planning scheme based on a distributed search of heuristic information and on a constraint programming resolution process. The overall approach is a distributed CSP resolution for solving the kind of problems that fit well a multi-agent planning paradigm.

The remainder of this paper is structured as follows. We begin by defining the problem characteristics and showing...
a representative problem example. Then, we present a general overview of our approach and we describe the main two stages of the planning algorithm. Finally, we show some experimental results and we present our conclusions and future work.

**Problem definition**

Our approach is particularly aimed at solving problems which develop in inherently distributed domains. In this type of problems, agents are clearly identified so it is not necessary to apply a problem decomposition because there exists a natural partition/distribution of the problem itself.

**Definition 1.** An agent is an entity with planning capabilities and thus we can specify an agent as a tuple $Ag = (Adj, G, I, A, m)$ where:

- $Adj$ is the set of adjacent agents. An agent $ag'$ is adjacent to an agent $ag$, $ag', ag \in Adj(ag)$, if the public information of $ag'$ is accessible from $ag$. This relationship is symmetric: $ag' \in Adj(ag) \iff ag \in Adj(ag')$. An agent can only collaborate directly with its adjacent agents, consulting and/or modifying their shared information.
- $G$ is the set of goals of the agent. These goals are not visible from other agents.
- $I$ is a set of propositions that represents the agent’s beliefs. This knowledge is classified in private (non-sharable) and public (sharable): $I = (Ip, Is)$. The private information, $Ip$, is a set of propositions that are not accessible from other agents. On the contrary, interactions between agents are possible through their public repository: other agents can consult and add propositions to $Is$.
- $A$ is the set of actions that can be applied in the domain. This information, which represents the agents’ skills, is not accessible from other agents. As a typical planning action, an action $a$ is a triple $(pre(a), add(a), del(a))$, where the preconditions, $pre(a)$ and the effects, $add(a)$ and $del(a)$, are sets of propositions. These propositions can be agent beliefs (public or private) or public beliefs from adjacent agents:
  \[
  \forall p \in \text{pre}(a) \cup \text{add}(a) \cup \text{del}(a) / a \in A(ag),
  \]
  \[
  p \in I(ag) \lor p \in Is(ag') : ag' \in Adj(ag)
  \]
- $m$ is the optimization function (or metric). The agent must try to achieve its goals with the minimum cost according to this function.

**Definition 2.** A planning problem consists of finding a (partially ordered) sequence of actions that leads the system from its initial state to a goal state. Formally, it is defined as a tuple $(Ag, I, G, A)$, where:

- $Ag$ is a set of agents.
- $I$ is the problem initial state, which is the union of the agents’ beliefs:
  \[
  I = \bigcup_{ag \in Ag} I(ag)
  \]
  This initial state is globally consistent because information is not replicated in different agents:
  \[
  \forall ag, ag' \in Ag, I(ag) \cap I(ag') = \emptyset
  \]

- $G$ is the problem goal, which is the union of the individual goals of all agents:
  \[
  G = \bigcup_{ag \in Ag} G(ag)
  \]
- $A$ is the problem actions, which is the union of the individual actions of all agents:
  \[
  A = \bigcup_{ag \in Ag} A(ag)
  \]

**Definition 3.** A global plan is a partially ordered set of pairs $(ag, a)$, where $ag$ is an agent and $a$ is an action of that agent ($a \in A(ag)$). The execution of $a$ by $ag$ causes a transition in the $ag$’s state and, possibly, in the public beliefs of adjacent agents. Thus, a global plan is valid if the execution of all actions in the plan (by their corresponding agents and respecting the ordering constraints) on the initial state leads to a final state where all problem goals hold.

The cost of a global plan is computed as the sum of the cost of all actions in the plan, according to the metric function defined in their corresponding agent. Therefore, metric functions of all agents must be defined by the same measure unit and the same scale. This simplification facilitates the computation of the plan quality, but it may not be adequate for certain problems where the optimization function of one agent gets into conflict with the objective of another agent. This problem will be addressed in a future work to improve the applicability of our problem model.

There are many real problems that fit the distributed problem paradigm we have described in this section. In the following subsection, we show a simple example that we will use through the rest of the paper to illustrate our proposal.

**Problem example**

In this section, we show a simple example from a transportation and storage problem. In this problem we can identify two types of agents: warehouses and transport companies.

In each warehouse, there is a storage area and a loading area. In the storage area, packages can be stacked and unstacked onto a fixed set of pallets by means of a hoist. All the information about the storage area (packages, stacks, ...) in a warehouse is private, so it is not visible from other agents. Packages can also be moved from/to the loading area. The loading area of a warehouse is public, so packages in that area are visible from the adjacent agents of the warehouse.

In general, the goal of a warehouse is to get a certain package distribution inside its storage area, and the metric function is to minimize the number of stack, unstack and move operations carried out to achieve its goal.

A transport company manages the transportation of packages between a set of warehouses located in the same geographic area. All the information about the truck fleet and the set of routes they use is private. Transport companies access the loading area of the warehouses to pick up and deliver packages. The optimization function in these companies is to minimize the total number of truck movements.

There are three agents in our example: two warehouse agents, $w1$ and $w2$, and a transportation agent $t1$. In the initial state, agent $w1$ has tree packages ($a, b$ and $c$) and agent $w2$ has two ($e$ and $d$), organized as Figure 1 shows. The
transportation agent has only one truck (\textit{truck1}) and links both warehouses. Warehouse agents are only adjacent to the transportation agent, so warehouses cannot communicate information among themselves.

We use the following predicates in this example:

- \texttt{on ?pkg1 ?pkg2}, which states that package \texttt{pkg1} is on package \texttt{pkg2}.
- \texttt{at-la ?pkg}, which indicates that package \texttt{pkg} is in the loading-area of the warehouse.

The goal of agent \textit{w1} is to get the package \textit{b} on top of package \textit{e}, that is, \texttt{on b e}. The position of the remaining packages in the warehouse is not relevant. The goals of agent \textit{w2} are \texttt{on d a} and \texttt{on a c}. Agent \textit{tl} has no individual goals.

The multi-agent planning approach that we propose in this paper can help agents in situations such as described in this example. It offers a way to co-operate while being in control of which information is shared and with whom.

**Distributed planning scheme**

Collaboration is required as agents usually cannot reach their individual goals without the help of the other agents. This help is provided through a set of abstract operations which we call \textit{services}.

**Definition 4.** A \textit{service} is an abstract operation that an agent offers to another agent to fulfill a set of public propositions, which we call the \textit{service goal}. Internally, a service is a (partially ordered) set of actions (local plan), but how an agent provides a service is kept as private information.

Additionally, we define an \textit{internal service} as an abstract operation that an agent computes for achieving its individual goals (in this case, the propositions in the service goal can be private). From now on, when we use the term service, we will refer to both services and internal services.

For providing a service, an agent may require the execution of one or more services from other agents. This way, the global planning problem consists of finding a partially ordered set of services that allows all agents to achieve their individual goals. The distributed planning process is started by the agents with individual goals to accomplish. The first step for these agents is to find out the pieces of information, which we call \textit{requirements}, that they require from other agents in order to achieve their goals.

**Definition 5.** A \textit{requirement} is a public proposition or a disjunction of public propositions that an agent needs from other agents to achieve its goals or to provide a service.

This way, a service can be seen as a planning operator, where the service requirements correspond to the operator preconditions and where the service goal corresponds to the operator effects.

In the proposed example, agent \textit{w2} has to stack package \textit{d} on top of \textit{a} and package \textit{a} on top of \textit{c}, but packages \textit{a} and \textit{c} are not in the warehouse. Therefore, the only way to achieve its goals is to get packages \textit{a} and \textit{c} in its loading area. Then, the requirements are \texttt{at-la a} and \texttt{at-la c}.

In the proposed example, requirements are always single propositions, but in other problems they can be disjunction of propositions. If, for instance, warehouse \textit{w2} had two different loading areas (\texttt{la1} and \texttt{la2}), then the requirements for reaching its goals would be: \texttt{(at-la1 a \lor at-la2 a)} and \texttt{(at-la1 c \lor at-la2 c)}.

Agents with requirements to fulfill have to ask their adjacent agents for help. The first stage of the planning algorithm is a message exchange process to find out what services agents can provide and for computing an estimated cost of these services. Once the set of available services has been established, the global problem is solved through a collaborative planning process, which is modeled as a distributed CSP. Both planning stages are described in detail in the following two sections.

**Cost estimate of the services**

In this stage, agents send messages to their adjacent agents to request services for satisfying their requirements. An agent that needs a proposition sends a service request message to its adjacent agents asking for the cost of that service. Only one proposition is requested in a single message. If a requirement contains disjunctive propositions, then these propositions are individually requested in separated messages. These service request messages contain the following information:

- \texttt{a_{orig}}: the requesting agent.
- \texttt{a_{dst}}: the target agent.
- \texttt{p}: the requested proposition.
- \texttt{R}: the message route, which is the sequence of agents the message has passed by (the target agent is not included). This information avoids infinite message loops.
- \texttt{Id}: the message identifier. Each time a message is propagated, a number is added at the end of the \texttt{Id}. This number is the same for all propositions in a requirement (that is, for disjunctive propositions) but different for propositions in different requirements.

Let’s suppose that an agent has the following requirements: \texttt{(p1 \lor p2)} and \texttt{p3}. The message identifier, for example, will be "1" for requesting \texttt{p1} and \texttt{p2} (in separated messages), and "2" for requesting \texttt{p3}. Through the message identifier and the message route parameters, an agent that receives several messages can easily find out if all the
requested propositions are required or if some of them are disjunctive, that is, alternative ways to satisfy a requirement. 

When an agent that receives a service request cannot provide the service, it returns an infinite cost as a reply. Otherwise, the agent must:

- Analyze the necessary requirements to provide the service.
- Ask its adjacent agents for help if required.
- If the requirements can be achieved, compute a plan to estimate the cost of the service. This cost is sent back as reply to the service request.

Additionally, if an agent receives two single different messages with non-disjunctive requested propositions, it automatically creates a new service for achieving both propositions together. This combination of propositions is done because achieving several propositions altogether is usually less costly than handling each subgoal independently and thus it will positively affect the plan quality.

However, since it is not affordable to compute all possible combinations, we will only consider the number of 2-combinations from a set with \( n \) non-disjunctive propositions \( (n^2(n-1)/2) \) plus one \( k \)-combination for \( k > 2 \). This calculation makes about \( n^2/2 \) number of combinations to study; this relaxation provides a good trade-off between computational cost and quality.

This message exchange process requires the agents to keep some information about the received messages and the requested and provided services. For each agent, the stored information is the following:

- **Messages database (MsgDB):** in this database the agent stores the received service request messages (the format of these messages was described above).

- **Service database (SerDB):** it stores the offered services. Tuples are in the form \( \langle G, a_{\text{orig}}, \text{NecReq}, \text{MinReq}, \text{minReqCost}, \text{Plan}, \text{cost} \rangle \), where:
  
  - \( G \) is the service goal, that is, the conjunction of propositions the service achieves.
  - \( a_{\text{orig}} \) is the agent that requested the service.
  - \( \text{NecReq} \) is the set of requirements needed to provide the service.
  - \( \text{MinReq} \) is a conjunction of propositions that satisfy the service requirements (\( \text{NecReq} \)) with the minimum cost. For each requirement with disjunctive propositions, the alternative with the minimum cost is selected.
  - \( \text{minReqCost} \) is the estimated cost to achieve \( \text{MinReq} \).
  - \( \text{Plan} \) is the internal plan that allows to achieve \( G \), assuming that the requirements hold.
  - \( \text{cost} \) is the estimated service cost, which corresponds to the \( \text{Plan} \) cost, computed according the metric function of the agent.

- **Requirements database (ReqDB):** this database stores the service request replies, that is, the cost of the services requested to other agents. This information is stored in tuples of the form \( \langle G, \text{cost}, a_{\text{dst}} \rangle \), where:
  
  - \( G \) is the service goal.
  - \( \text{cost} \) is the service cost, that is, the estimated cost to achieve the propositions in \( G \).
  - \( a_{\text{dst}} \) is the agent that provides this service.

Algorithm in Figure 2 shows the behaviour of agent \( a_{\text{dst}} \) when it receives a service request from agent \( a_{\text{org}} \). In this algorithm, we have used the following functions:

- \( \text{Unreachable}(p) \) returns \text{true} if no action allows the agent to achieve proposition \( p \), regardless of the preconditions of that action holds or not.
- \( \text{ComputeNecessaryRequirements}(p) \) returns the necessary requirements to achieve \( p \).
- \( \text{RequirementsAchieved}(\text{NecReq}) \) returns \text{true} if \( \text{NecReq} \) can be satisfied.
- \( \text{ComputeMinimumRequirements}(\text{NecReq}) \) returns the set of propositions that satisfies \( \text{NecReq} \) with the minimum cost.
- \( \text{ComputePlan}(S, G) \) returns a plan to achieve the set of propositions \( G \) from the initial state \( S \).
- \( \text{PlanCost}(\text{Plan}) \) returns the cost of \( \text{Plan} \) according to the defined problem metric.
- \( \text{Conjunctive}(p_1, p_2) \) returns \text{true} if both propositions are conjunctive. This can be easily computed through the route and the identifier of their respective messages.
- \( \text{ServiceCombination}(G) \) method computes the cost of a service that achieves all propositions in \( G \). Therefore, the minimum requirements and a new plan to achieve \( G \) must be computed. If this service is less costly than achieving all propositions in \( G \) separately, then a message containing this information is delivered to the requesting agents.

At the end of this process, each agent has a list of services with an estimated cost for each one of them. Since the purpose of these services is to help other agents achieve their requirements, each agent must compute a set of internal services to achieve its own goals. Agent \( w_1 \), for example, will compute an internal service for satisfying its goal ‘on b e’. Combinations with other services are also studied, in the same way as shown in Figure 2.

Tables 1, 2 and 3 show the services offered by agents \( w_1 \), \( w_2 \) and \( n_1 \) respectively. The first column assigns a number to each service. The second column shows the service goal: each proposition is preceded by the requesting agent. The third column indicates the service cost, without taking into account the cost of the requirements. The last column shows the necessary requirements to provide the service: each proposition is preceded by the agent that can achieve it with the minimum cost.

**Collaborative planning**

In the literature we can find some proposals for solving collaborative planning tasks (Cox, Durfee, & Bartold 2005) (Rehak, Pechoucek, & Volf 2006). These works address
a different problem than ours, since they follow a divide-and-conquer approach, but the key idea is the same: using POP (Partial Order Planning) techniques (Penberthy and Weld 1992). POP techniques are very appropriate for distributed planning since no explicit global state is required.

Instead of developing a new distributed POP algorithm, we have chosen to convert the planning problem into a distributed constraint satisfaction problem (disCSP). The reasons for this decision are:

- There are many distributed CSP algorithms (Yokoo and Hirayama 2000) and some available platforms, such as DisChoco 1.
- A planning problem can be easily formulated as a CSP and experimental results show that this approach can be very competitive (Vidal & Geffner 2006).

The goal of this process is to obtain a final global plan, which is a partially ordered list of services that each agent will have to carry out. In order to establish the order between these services, each service will have an associated starting time. The steps for obtaining this global plan are described in the following subsections.

---

1Available at http://www.lirmm.fr/coconut/dischoco/

**Selecting the agent priority**

In this type of decentralized algorithms is necessary to establish an order/priority between the participating agents. Moreover, this order substantially affects the performance of the search.

Experimentally, we have observed that the most efficient order assignment is to set the highest priority to the agent with the lower number of services (internal services are not considered). In the proposed example, $w_2$ is the agent with highest priority since it only has one non-internal service: the service #1. The rest of services are internal.

The next agent in the priority order is the agent that provides a higher number of services to agents with an already assigned priority, and so on. In the proposed example, $t_1$ is the agent that provides more services to $w_2$. Finally, $w_1$ will be the agent with the lowest priority.

---

### Table 1: List of services of agent $w_1$.  

<table>
<thead>
<tr>
<th>#</th>
<th>Service goal</th>
<th>Cost</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${t_1:\text{at-la a}}$</td>
<td>6</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>2</td>
<td>${t_1:\text{at-la c}}$</td>
<td>2</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>3</td>
<td>${t_1:\text{at-la a}, t_1:\text{at-la c}}$</td>
<td>6</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>4</td>
<td>${w_1:\text{on b e}}$</td>
<td>6</td>
<td>${t_1:\text{at-la e}}$</td>
</tr>
<tr>
<td>5</td>
<td>${t_1:\text{at-la a}, w_1:\text{on b e}}$</td>
<td>8</td>
<td>${t_1:\text{at-la e}}$</td>
</tr>
<tr>
<td>6</td>
<td>${t_1:\text{at-la c}, w_1:\text{on b e}}$</td>
<td>6</td>
<td>${t_1:\text{at-la e}}$</td>
</tr>
<tr>
<td>7</td>
<td>${t_1:\text{at-la a}, t_1:\text{at-la c}, w_1:\text{on b e}}$</td>
<td>8</td>
<td>${t_1:\text{at-la e}}$</td>
</tr>
</tbody>
</table>

### Table 2: List of services of agent $w_2$.  

<table>
<thead>
<tr>
<th>#</th>
<th>Service goal</th>
<th>Cost</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${t_1:\text{at-la e}}$</td>
<td>2</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>2</td>
<td>${w_2:\text{on d a}}$</td>
<td>6</td>
<td>${t_1:\text{at-la a}}$</td>
</tr>
<tr>
<td>3</td>
<td>${t_1:\text{at-la e}, w_2:\text{on d a}}$</td>
<td>6</td>
<td>${t_1:\text{at-la a}}$</td>
</tr>
<tr>
<td>4</td>
<td>${w_2:\text{on a c}}$</td>
<td>4</td>
<td>${t_1:\text{at-la a}, t_1:\text{at-la c}}$</td>
</tr>
<tr>
<td>5</td>
<td>${w_2:\text{on a c}, w_2:\text{on d a}}$</td>
<td>8</td>
<td>${t_1:\text{at-la a}, t_1:\text{at-la c}}$</td>
</tr>
<tr>
<td>6</td>
<td>${t_1:\text{at-la e}, w_2:\text{on a c}}$</td>
<td>6</td>
<td>${t_1:\text{at-la a}, t_1:\text{at-la c}}$</td>
</tr>
<tr>
<td>7</td>
<td>${t_1:\text{at-la e}, w_2:\text{on a c}, w_2:\text{on d a}}$</td>
<td>8</td>
<td>${t_1:\text{at-la a}, t_1:\text{at-la c}}$</td>
</tr>
</tbody>
</table>

### Table 3: List of services of agent $t_1$.  

<table>
<thead>
<tr>
<th>#</th>
<th>Service goal</th>
<th>Cost</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${w_1:\text{at-la e}}$</td>
<td>2</td>
<td>${w_2:\text{at-la e}}$</td>
</tr>
<tr>
<td>2</td>
<td>${w_2:\text{at-la a}}$</td>
<td>1</td>
<td>${w_1:\text{at-la e}}$</td>
</tr>
<tr>
<td>3</td>
<td>${w_1:\text{at-la e}, w_2:\text{at-la a}}$</td>
<td>2</td>
<td>${w_2:\text{at-la e}, w_1:\text{at-la a}}$</td>
</tr>
<tr>
<td>4</td>
<td>${w_2:\text{at-la c}}$</td>
<td>1</td>
<td>${w_1:\text{at-la c}}$</td>
</tr>
<tr>
<td>5</td>
<td>${w_1:\text{at-la e}, w_2:\text{at-la c}}$</td>
<td>2</td>
<td>${w_2:\text{at-la e}, w_1:\text{at-la c}}$</td>
</tr>
<tr>
<td>6</td>
<td>${w_2:\text{at-la a}, w_2:\text{at-la c}}$</td>
<td>1</td>
<td>${w_1:\text{at-la a}, w_1:\text{at-la c}}$</td>
</tr>
<tr>
<td>7</td>
<td>${w_1:\text{at-la e}, w_2:\text{at-la a}, w_2:\text{at-la c}}$</td>
<td>2</td>
<td>${w_1:\text{at-la a}, w_1:\text{at-la c}}$</td>
</tr>
</tbody>
</table>
The reasons for these ordering criteria are the following:

- The agents with a higher number of services and, consequently, with a greater number of variables and constraints, have low priorities. This way, the agents that have to make a greater computation effort are the last in the CSP solving process, thus minimizing the number of required backtracks.

- Following an assignment order according to the services’ causal links, we maximize the number of shared variables and constraints between two consecutive agents. Then, when an agent communicates its partial solution to the next agent, it is possible to prune the domains of the variables efficiently.

Formulating the problem as a CSP

Each service can be easily translated into a PDDL operator: the requirements correspond to the operator’s preconditions and to the delete effects (if they do not hold after the service execution), the service goals correspond to the add effects, and the service cost can be modeled as a numeric fluent. For example, the third service of agent $t1$ (see Table 3) is translated as follows:

\[
(:\text{action} \; t1-\text{Service3} \\
:\text{parameters}() \\
:\text{precondition} \; \text{(and} \; (w2-\text{at-la} \; e) \; \; (w1-\text{at-la} \; a) \; \text{)} \\
:\text{effect} \; \text{(and} \; \text{(not} \; (w2-\text{at-la} \; e)) \; \; \text{(not} \; (w1-\text{at-la} \; a)) \; \; (w1-\text{at-la} \; e) \; \; (w2-\text{at-la} \; a) \; \; \text{(increase} \; \text{(cost} \; 2)))\]

Based on the works of (Vidal & Geffner 2006) and (Refanidis 2005), we can translate these planning operators/services into a CSP formulation. For this formulation, we have defined the following integer variables:

- $\text{Inplan}(o) \in [0,1]$, represents whether the operator $o$ is in the final plan (value 1) or not (value 0).

- $\text{Start}(o) \in [0,\infty]$, represents the start time of $o$.

- $\text{Support}(p, o) \in O$, where $O$ is the set of operators that can provide proposition $p$ for $o$. Thus, these variables represent the causal links between the agent services.

- $\text{Time}(p, o) \in [0,\infty]$, is the time when the causal link $\text{Support}(p, o)$ happens.

If an operator $o$ is included in the plan ($\text{Inplan}(o) = 1$), then the following constraints must hold:

- The operator that produces $p$ for $o$ must be in the plan: $\text{Support}(p, o) = o' \implies \text{Inplan}(o') = 1$.

- Preconditions of $o$ must hold before its start time: $\text{Time}(p, o) \leq \text{Start}(o)$.

- The duration of an operator is one time unit (we are not working with durative actions), so its effects are achieved one unit of time after the operator’s start: $\text{Support}(p, o) = o' \implies \text{Time}(p, o) = \text{Start}(o') + 1$.

- $\text{Mutex}$ relationship between an operator $o$ that requires $p$ and other operator $o'$ in the plan that deletes $p$: $\text{Start}(o) \neq \text{Start}(o') + 1$.

- Threat resolution by promotion or demotion when an operator $o'$ in the plan deletes a proposition $p$ that is required by $o$: $(\text{Start}(o') + 1 < \text{Time}(p, o)) \lor (\text{Start}(o) < \text{Start}(o'))$.

Each agent is in charge of dealing with the variables and constraints that are related to its own services. However, there are some variables and constraints that must be shared between two agents. This occurs when the results of a service provided by an agent are required by other service of another agent. In this case, the agent with the highest priority will assign a value to the shared variables, whereas the other agent will check the shared constraints.

Heuristics

The formulation of a planning problem as a CSP requires the definition of a great number of variables and constraints. For instance, there are 227 variables and 781 constraints in the proposed example (57, 86 and 84 variables and 182, 297 and 302 constraints for agents $w2$, $t1$ and $w1$, respectively). In order to improve the efficiency of the CSP solver, we have defined some additional constraints and a specific value selection heuristic for the $\text{Inplan}$ variables.

One reason for the problem complexity is the high number of services that arise from the requirement combinations: if two conjunctive propositions, $p1$ and $p2$, have been requested to an agent, that agent automatically computes a new service that jointly achieves $p1$ and $p2$. Evidently, if the service to achieve $p$ is included in the plan, the service to achieve $p1 \land p2$ will not be included (and the other way around), since both services represent alternative ways to attain $p$. To model this fact in the CSP formulation, we have included the following constraints: if two operators in an agent, $o$ and $o'$, produce a proposition $p$, then both operators are mutually exclusive in the plan: $(\text{Inplan}(o) = 1 \implies \text{Inplan}(o') = 0) \land (\text{Inplan}(o') = 1 \implies \text{Inplan}(o) = 0)$. These additional constraints substantially improve the solving process performance.

The most costly part for the CSP solver is to find out what operators must be included in the plan, that is, to make a feasible value assignment for the $\text{Inplan}$ variables. For this reason, each agent tries to make a good initial value assignment for these variables:

- A set with the requested propositions is computed for each adjacent agent. The agent itself is also considered as an adjacent agent as internal services can be seen as self-requests.

For agent $w2$ in the proposed example, these sets are the following:

- \text{Requests of agent } t1: \{at-la e\}
- \text{Self-requests: } \{on d a, on a c\}

- A set of services/operators is computed to satisfy each set of requested propositions with the minimum cost. Following the example, these sets of services for agent $w2$ are the following:
Distributed solving process

In the literature we can find several distributed CSP algorithms: asynchronous backtracking (ABT), asynchronous weak-commitment search (AWC), distributed breakout, etc. (Yokoo & Hirayama 2000). However, we are not interested at present in solving the problem the most efficiently as possible, but in demonstrating that our approach is viable for solving this type of problems. For this reason, we have implemented a simple sequential forward checking algorithm. As a future work, we want to use a more sophisticated algorithm in order to decrease the number of messages that the agents need to exchange during the search.

Plan execution

At the end of the process each agent knows what services have to execute and at what time. Table 4, for instance, shows the final plan obtained for the proposed example. This way, the plan execution becomes an easy task.

However, the final plan might not be completely executable in certain cases. This is because the computation of the estimate cost of the services is based on the problem initial state, which changes through the plan execution. Therefore, it is possible that some services that were initially available cannot be executed later (after the execution of other services).

If an agent must execute a service according to the global plan, but it is not possible to achieve the goals of that service in the current state, then it is necessary to calculate a new global plan: the planning process is repeated again but starting from the current situation. This solution does not avoid the problem of dead-ends, which may appear in some non-reversible domain. This issue will be addressed in a future work.

Results

For the evaluation of our approach, we have defined four problem examples with an increasing number of agents. The first problem, with three agents, corresponds to the proposed problem example (see Figure 1). The second problem adds a new warehouse, \textit{w3}, with an individual goal to achieve. The third problem adds a new transport company, \textit{t2}. The last problem, with six agents, includes a new warehouse, \textit{w4} with another individual goal to solve.

The obtained results are displayed in Table 5. The first column indicates the number of agents in the problem. The second column shows the total number of services provided by the agents. The third and the fourth column show the total number of variables and constraints in the CSP formulation of the problem, respectively. The fifth column shows the number of messages exchanged during the CSP solving process. The last column shows the time in seconds for obtaining the first solution. The distributed CSP solver has been executed on a single computer (Pentium 4 - 3Ghz.), so we have not considered message delays for the agents communication.

The goal of this evaluation is not to show a broad range of benchmark results (since the whole process is not fully automated yet), but to demonstrate that our approach is valid for solving this type of problems. In spite of the current implementation can be improved in many ways, the low number of messages exchanged and the small computation times indicate that our approach can be successfully used for solving small and medium multi-agent planning problems.

Thanks to the value selection heuristics, the plan quality obtained in the first solution found is usually very good. In the proposed example, for instance, the first solution found is the optimal one. In the current implementation of the CSP solver, the search is stopped when the first plan is found. However, this behaviour can be easily changed to allow the search to continue until a deadline is reached or even until the optimal solution is found.

Conclusions and future work

In this paper, we have presented a new approach for multi-agent planning, based on the extraction of heuristic information and the problem formulation as a CSP. Unlike other existing proposals that follow a divide-and-conquer approach, we focus our work on inherently distributed problems in which agents are clearly identified. In this type of problems, privacy is usually a key issue: agents keep private information that they do not want to share with other agents. At the
same time, some information must be shared to allow the cooperation between the agents, which is required to reach their goals.

In our approach, the collaboration and the privacy between agents is achieved through the definition of services, which are abstract operations provided by the agents. This way, the planning process consists of two sequential steps:

- A message exchange process where agents request the information that they need to other agents and where the set of available services is obtained. The cost of each service is used as heuristic information for the search process.
- The problem is formulated as a distributed constraint satisfaction problem and solved with CSP techniques.

Some preliminary results show that our approach can be successfully used for solving small and medium size problems. Moreover, the quality of the solution is usually very good.

As a future work, we want to fully automatize the planning process, using a multi-agent platform such as, for example, JADE (Bellifemine, Caire, & Greenwood 2007). This will allow us to test our algorithm in a wide range of benchmark domains. On the other hand, there are some improvements that can be introduced in the algorithm, such as the use a more efficient distributed CSP solving method and the computation of new heuristics for the variable and the value selection. Finally, there are two key points that will be addressed in future works to improve the applicability and utility of our proposal: the issue of the global plan quality, allowing to define conflicting optimization functions for the agents, and the prevention of possible dead-ends during the planning process.

**Acknowledgements**

This work has been partially supported by the Spanish government project MCyT TIN2005-08945-C06-06 (FEDER) and by the Valencian government project GV06/096.

**References**


Plan Coordination for Durative Tasks

J. Renze Steenhuisen and Cees Witteveen
Faculty of Electrical Engineering, Mathematics and Computer Science,
Delft University of Technology
P.O. Box 5031, 2600 GA Delft, The Netherlands

Abstract
In multi-agent domains, agents can be given planning or scheduling autonomy through coordination. However, plan coordination discards available scheduling information, while schedule coordination possibly over-constrains the problem from a planning point of view. In an attempt to attack this problem, we included more temporal information used in schedule coordination into the plan-coordination framework. We discovered that plan coordination can be achieved when reasoning with qualitative temporal constraints and durative tasks. Additionally, we defined this new coordination problem and studied its complexity. The relevance of this work is that it studies plan coordination in a wider context, such that the agents can—through coordination—be provided with both planning and scheduling autonomy.

Introduction
Agents are being introduced in a wide variety of task domains, because they promise to increase agility. These agent systems are emerging in such diverse domains as multimodal transportation (Chiu et al. 2005), firefighting with unmanned aerial vehicles and air traffic control (Léauté & Williams 2005), and crisis response (Harrald 2005). In general, the tasks that need to be performed in these domains are interdependent, require more than one agent to execute them, require a task-planning process for every individual agent, and need to be completed on time. Obviously, due to the task interdependencies, some form of coordination mechanism is needed to ensure that the results of the task-planning processes are jointly feasible. In general, such a coordination mechanism (Christodoulou, Koutsoupias, & Nanavati 2004) should enable individual agents to choose their preferred way to solve their part of the task, thereby (minimally) reducing the initial (planning) autonomy of the agents. Therefore, the quality of a coordination mechanism depends on both the severity of the restrictions imposed and the overall performance quality it allows. In this paper, we propose and analyse the (computational complexity) properties of such a coordination mechanism for interrelated tasks with time constraints. Before we introduce the basics of our coordination framework, we will place it in a more general perspective.

Background
First of all, the reader should be aware that there are several existing approaches to solve this plan-coordination problem. For example, in (Smith et al. 2007) the authors propose to manage the coordinated planning and execution of the tasks by letting the agents keep each other informed about any changes (e.g., completed, new, or re-scheduled tasks). It is clear that such an approach requires the agents to be collaborative, each agent willing to inform other agents about details of its individual plan and cannot be used if the agents are competitive or self-interested. Moreover, this mechanism will fail when communication is impossible or difficult to establish.

In this paper, we advocate a more principled approach to the plan-coordination problem. First of all, we distinguish different phases in a multi-agent planning process each requiring some form of coordination. Second, we distinguish some specific interrelationships between tasks and agents that determine which form of coordination is required in the (multi-agent) planning process. Together, these phases and interrelationships determine which form of coordination is required in which phase of the planning process (see also (Zlot & Stentz 2006)).

To start with the four main phases in the planning process, in the allocation phase, tasks are assigned to agents that are capable of completing it. Second, the order in which the tasks are to be executed is determined in the planning phase. Third, a time schedule is constructed for the tasks in the scheduling phase that is compatible with the plan. Last, we have the execution phase in which the tasks are executed according to the constructed schedule.

The interrelationship between tasks and agents determine in which phase plan coordination is needed and which form of plan coordination should be applied. First, a set of tasks \( M \) is called loosely coupled with respect to a set of agents \( \mathcal{A} \) when the tasks occurring in \( M \) can be assigned to the agents \( A \in \mathcal{A} \) such that there exist no dependency relations between tasks assigned to different agents. Clearly, if we have a loosely-coupled system, each agent is able to construct an independent plan for its subset of tasks. Therefore, coordination is not needed in the planning, scheduling, and execution phase and reduces to solving a task-allocation problem. Examples of this category are tasks that are totally independent of each other, such as searching for casualties in different parts of a city.
Second, if the set of tasks $M$ is partially ordered by some dependency relation, and it is impossible to assign the tasks such that tasks assigned to different agents in $A$ have no dependency relation between them, then $M$ is said to be moderately coupled. For these complex tasks, plan coordination is required before or during the planning phase in order to ensure that the induced partially-ordered dependency relation between agents is preserved by the individual planning processes. There is, however, no a-priori need to provide a coordination mechanism in the scheduling or execution phase if plan coordination can be guaranteed. Typical problems in this category are monitoring tasks, patient scheduling, and multi-modal transportation tasks.

Finally, if the set of tasks $M$ is moderately coupled and, moreover, requires the satisfaction of constraints (e.g., time constraints) when scheduling and executing the tasks, the set of tasks is said to be tightly coupled. Here, coordination is required in the planning, scheduling, and execution phase. Examples of tightly-coupled tasks are (i) extinguishing a large fire that requires simultaneous action of multiple firefighters from different angles, and (ii) simultaneously lifting a patient onto a bed.

**Plan coordination: previous and current work** In previous work (Steenhuisen et al. 2006; Buzing et al. 2006), we concentrated on the the plan-coordination problem for moderately-coupled tasks. Basically, in these papers, we showed that there exists a plan-coordination mechanism (although difficult to design) that is able to reduce a moderately-coupled task to a loosely-coupled task. In other words, such a plan-coordination mechanism is able to guarantee the participating agents that they can plan completely independent from the other agents while still guaranteeing the feasibility of the joint plan. This approach is robust against failing communication, allows individual replanning, and additional tasks to be done by the agents while not violating global constraints. Moreover, this approach is applicable in many other domains where agents are unwilling or unable to revise their plans.

In more recent work (Steenhuisen & Witteveen 2007), we have extended this approach to tightly-coupled tasks with dependency constraints and synchronisation constraints. We showed that there exist plan-coordination mechanisms that guarantee independent planning by the individual agents, but require information exchange after planning to establish the exact time of scheduling synchronisation tasks.

In this paper, we will extend the planning approach, showing that we can also provide coordination mechanisms for durative tasks with time constraints. The coordination mechanisms developed ensure the participating agents that whatever feasible plan they provide for their own set of tasks, there always exists a schedule for the joint plan obtained by assembling the individual plans. However, this does not mean that all locally feasible schedules are guaranteed to form a feasible joint schedule. In this paper, we reduce the construction from previously developed coordination mechanisms to those with durative tasks and time constraints, which implies that the latter are at least as hard to design as the former mechanisms.

**Framework**

We consider a set of agents $A = \{A_1, \ldots, A_n\}$ that have to complete a complex task $T$. Such a complex task consists of a set $M = \{m_1, \ldots, m_k\}$ of methods $m_i$, together with a set of constraints on the execution of these methods. These constraints can be partitioned into a set of dependency constraints, determining for each method the set of methods it is dependent upon and a set of synchronisation constraints determining which methods $m_j$ should be executed concurrently with a given method $m$.

For a uniform representation also suitable for temporal planning aspects, we use a set of time points $T$ and binary relations between them to represent methods and relations between them. First of all, each $m_i \in M$ is represented by an ordered pair $(t_i,s_i)$ where $t_i$ is the time point indicating the starting time of $m_i$ and $s_i$ its ending time. Dependencies and synchronisation relations between methods can now be represented by relations between time points as follows. We distinguish a partial order $\prec \subseteq (T \times T)$ representing the dependency relation and an equivalence relation $\equiv \subseteq (T \times T)$ for synchronisation. More precisely, if a method $m_j$ depends on method $m_i$, there is a dependency constraint $t_i \prec t_j$. Furthermore, for every method $m_i$, $t_i \prec t_i$. If $m_i$ is synchronised with $m_j$, then $t_i \equiv t_j$. Note that since $\equiv$ is an equivalence relation, each time point $t \in T$ is synchronised with itself.

Finally, the composition of $\prec$ and $\equiv$ satisfies the following two natural inclusion properties.

1. $(\prec \circ \equiv) \subseteq \prec$ and $(\equiv \circ \prec) \subseteq \prec$, which means that $(t \prec t' \land t' \equiv t'' \text{ implies } t \prec t''$) and $(t \equiv t' \land t' \prec t'' \text{ implies } t < t'')$, and
2. $(\prec \cap \equiv) = \emptyset$, meaning that $\prec$ and $\equiv$ are orthogonal relations.

**Temporal relations and classification of complex tasks**

In many planning domains (e.g., airport planning, manufacturing, and supply-chain management), we have to account for temporal constraints that constrain the execution of a method relative to the execution of other methods. Seven\(^1\) of such qualitative temporal constraints have been identified for (qualitative) time intervals: before, overlaps, during, meets, starts, finishes, and equals (Allen 1983). In previous work (Steenhuisen & Witteveen 2007), we showed that all these qualitative temporal constraints can be represented in a task framework with time points, together with the above precedence and synchronisation constraints.

Also, a clear distinction is emerging on the degree to which methods can be coupled using these constraints. On the one hand, we have the constraints that require both precedence and synchronisation constraints (i.e., meets, starts, finishes, and equals), while others only need precedence constraints (i.e., before, overlaps, and during). We call the complex tasks in which the end points of methods are constrained by both precedence and synchronisation constraints tightly-coupled tasks, when only prece-

\(^1\)Neglecting the converse of each of these relations.
Plan coordination for moderately-coupled tasks

Note that in moderately-coupled tasks, we have a set of time points $T$ and a partially-ordered precedence relation $\prec$ between time points. Each method $m_i \in M$ is represented by an ordered pair $(t_{i,e}, t_{i,s})$ of time points. We assume the original set $M$ to be partitioned into $n$ disjoint sets $M_i$, representing the subset of methods to be executed by agent $A_i$. Therefore, the complex task assigned to $A_i$ can be represented by a partial order $\langle T_i, \prec_i \rangle$, where $T_i$ is the subset of time points associated with $M_i$, and $\prec_i$ the partial order restricted to $T_i$.

The agents are assumed to be independent and self-interested planners, able to reason with all given information. Therefore, every agent $A_i$ is allowed to come up with an individually chosen plan, ordering the methods it received, as long as it is compatible with the original constraints $\prec_i$. That implies that every plan $\langle T_i, \prec_i^* \rangle$ could be put forward by an agent $A_i$ as long as $\prec_i^*$ is a partial order extending $\prec_i$.

It is easy to see that, in many cases, not every combination of such autonomously chosen plans will result in a jointly feasible plan. For example, in Figure 1(a), a complex task is shown where agents $A_1$ and $A_2$ can plan $m_4 < m_1$ and $m_2 < m_3$ (see Figure 1(b)). But when these plans are joined, a cycle $\langle m_1, m_2, m_3, m_4, m_1 \rangle$ is introduced. Such a cycle indicates an infeasible joint plan, since it implies $m_1$ to precede $m_2$, but also vice versa. Since such an infeasible combination of individually feasible plans is possible, we call this complex task uncoordinated.

Thus, a plan-coordination mechanism should ensure that whatever feasible plans are chosen by the individual agents $A_i$, the joining of these plans constitutes a feasible global plan for the original set of methods $M$, satisfying all dependencies. Such a plan-coordination mechanism then should prevent every potential inter-agent cycle.

Plan coordination for tightly-coupled tasks

Analogous to moderately-coupled tasks, the coordination problem for a tightly-coupled task $\langle \langle T_i \rangle_i, \prec, \equiv \rangle$ occurs if methods in some set $M$ are assigned to agents such that each agent $A_i$ has to complete a part $\langle T_i, \prec_i, \equiv_i \rangle$ of it. However, since some methods need to be synchronised, the agents want to find individually suitable plans that they do not need to revise when agreeing upon a joint schedule for the set of methods. The coordination problem for tightly-coupled tasks then is how to ensure that, whatever individually feasible plan is chosen for an agent’s complex task, there will always exist a joint schedule for the total set of methods that satisfies each of the individual plans.

Clearly, in order to complete his part $\langle T_i, \prec_i, \equiv_i \rangle$ each agent $A_i$ can choose a plan $\pi_i = \langle T_i, \prec_i^*, \equiv_i^* \rangle$ for it, where $\prec_i \subseteq \prec_i^*$ and $\equiv_i \subseteq \equiv_i^*$. Such a plan can simply be conceived as a refinement of the partially-ordered set $\langle T_i, \prec_i, \equiv_i \rangle$.

Plan coordination can be achieved by plan decoupling (Valk 2005): Adding precedence constraints to the set of time points of each agent such that every agent $A_i$ is allowed to autonomously construct a plan for its set $M_i$ of methods, respecting only the set $\prec_i$ of local constraints and the existence of a feasible joint plan is always guaranteed.

The plan-decoupling problem (PDP) is to find a minimum set of such additional constraints.

**PDP for Moderately-Coupled Tasks**

**INSTANCE:** A moderately-coupled task $\langle \langle T_i \rangle_i, \prec \rangle$ and a positive integer $K$.

**QUESTION:** Does there exist a coordination set $\Gamma$ with $|\Gamma| \leq K$ such that $\langle \langle T_i \rangle_i, (\prec \cup \Gamma)^+ \rangle$ is coordinated? 2

This problem, as well as some of its variants, has been studied quite extensively. It turns out that this problem is $\Sigma_2^P$-complete in general (Valk 2005), and NP-complete when the number of agents is bounded by some constant (Steenhuisen et al. 2006). In addition, it was shown that this decoupling problem is APX-hard, and that a constant-ratio approximation algorithm is not likely to exist (Valk 2005). There exists a simple polynomial-time algorithm that finds a sufficient—but not necessarily minimum—coordination set for distributed tasks with precedence constraints. For some restricted cases of plan coordination, this algorithm has even been shown to be a constant-ratio approximation algorithm (ter Mors, Valk, & Witteveen 2006).

---

2How to find a suitable assignment for a set of agents is a separate problem (Zlot & Stentz 2006; Shehory & Kraus 1998), and is beyond the scope of this paper.

3For any relation $\rho$, the transitive closure is denoted by $\rho^+$.
The joint plan for a set of agents $A$ on a tightly-coupled task $(T_i)_{i=1}^n$, $i \equiv$ is a plan $\pi = (T_i)_{i=1}^n, \equiv^+$ where

(i) each plan $\pi_i = (T_i, \equiv^+_i, \equiv^+_i)$ of agent $A_i$ is respected, i.e.,

$\equiv^+_i \subseteq (\equiv^+ \cap (T_i \times T_i))$, (ii) $\equiv^+_i \subseteq \equiv^+$, and (iii) $\equiv^+_i \equiv \equiv^+$.

An individual schedule $s_i : T_i \rightarrow \mathbb{Z}$ is said to satisfy the individual plan $\pi = (T_i)_{i=1}^n, \equiv^+$ of agent $A_i$ if the following conditions hold:

1. $\forall t, t' \in T_i; t < t'$ implies $s_i(t) < s_i(t')$, and
2. $\forall t, t' \in T_i; t \equiv t'$ implies $s_i(t) = s_i(t')$.

Finally, a set of individual schedules $s_i \in \{s_i\}_{i=1}^n$ of individual plans $\pi_i$ constitutes a joint schedule if the following holds:

1. $\forall t \in T_i, t' \in T_j; t < t'$ implies $s_i(t) < s_j(t')$, and
2. $\forall t \in T_i, t' \in T_j; t \equiv t'$ implies $s_i(t) = s_j(t')$.

Now, we say that a tightly-coupled task $(T_i)_{i=1}^n, \equiv^+$ assigned to a set of agents $A$ is coordinated if for every combination $\pi_i \in \{\pi_i\}_{i=1}^n$ of individually chosen feasible plans there exist a set of schedules $s_i \in \{s_i\}_{i=1}^n$ such that each $s_i$ satisfies $\pi_i$, and $\{s_i\}_{i=1}^n$ constitutes a joint schedule.

Analogous to the moderately-coupled case, coordination for tightly-coupled tasks can be achieved by plan decoupling, that is finding a (minimum) set of additional constraints that allow agents to plan autonomously while guaranteeing the existence of a joint schedule based on their individually chosen plans. The associated problem is the following plan-decoupling problem.

**PDP for Tightly-Coupled Tasks**

**INSTANCE:** A tightly-coupled task $(T_i)_{i=1}^n, \equiv^+$ and a positive integer $K$.

**QUESTION:** Does there exist a coordination set $\Gamma$ with $|\Gamma| \leq K$ such that $(T_i)_{i=1}^n, \equiv \cup \Gamma^+$ is coordinated?

Recently, we showed that PDP for tightly-coupled tasks is $\Sigma_2^P$-complete (Steenhuisen & Witteveen 2007). Furthermore, we showed that, with some minor modifications, the same approximation algorithm can be used for moderately-coupled tasks.

Surprisingly, viewing the problem computationally, coordinating moderately or tightly-coupled tasks does not differ significantly.

**Plan Coordination of Durative Task Networks**

In the previous section, we described our framework for studying PDP, and summarised some relevant achieved results. The most elaborate complex tasks on which PDP was studied, are the tightly-coupled tasks. Although the qualitative temporal constraints can be used in these tasks already, it is not possible to use any form of quantitative temporal constraint. In this section, we make a start at closing this gap by introducing both time windows on the methods and durations of methods to the framework.

We formally represent our extended tightly-coupled tasks as a tuple $(T_i)_{i=1}^n, \equiv, I, \delta$, and refer to it as a Durative Task Network (DTN). Here, the first three entries are equal to those used in tightly-coupled tasks. Then, for each time point $t$, there is a time window (or temporal interval) $I(t) = (lb(t), ub(t))$, with $lb(t) < ub(t)$, $lb(t) \in \mathbb{Z} \cup \{-\infty\}$ and $ub(t) \in \mathbb{Z} \cup \{\infty\}$. The most relaxed time window is the universal time window, $(-\infty, \infty)$, which bounds are used when no lower or upper bound is provided. Furthermore, each method $m_i$ has a certain fixed duration $\delta(m_i) \in \mathbb{Z}^+$ that represents the temporal distance from $t_{i,s}$ to $t_{i,e}$. Clearly, it must hold for method $m_i$ that $ub(t_{i,s}) - lb(t_{i,s}) \geq \delta(m_i)$. Note that a time window $[x, y]$ can be rewritten to $(x - \epsilon, y + \epsilon)$, where $\epsilon$ is the smallest temporal distance between two time points.

Without loss of generality, we assume the following intuitive properties to hold for each DTN:

- If $t < t'$ then $lb(t') := \max(lb(t) + \epsilon, lb(t'))$ and $ub(t') := \min(ub(t), ub(t') - \epsilon)$.
- If $t \equiv t'$ then $I(t) := I(t') := [\max(lb(t), lb(t')), \min(ub(t), ub(t'))]$, and
- for each method $m_i$, $lb(t_{i,s}) := \min(ub(t_{i,s}), ub(t_{i,e}) - \delta(m_i))$ and $lb(t_{i,e}) := \max(lb(t_{i,s}) + \delta(m_i), lb(t_{i,e}))$.

A DTN for which these conditions hold, is called normalised. Henceforth, we assume each DTN to be normalised unless stated otherwise. Note that the time windows of $t_{i,s}$ and $t_{i,e}$ have the same width, and that this width is the available slack. We need $\epsilon \leq \min_i \delta(m_i)$, but we can assume, without loss of generality, that the smallest temporal distance is $\epsilon = 1$. Others have already reported on great gains in practise for scheduling by using these tighter bounds (Sultanik, Modi, & Regli 2006).

A feasible schedule $s : T \rightarrow \mathbb{Z}$ for a DTN $(T_i)_{i=1}^n, \equiv, I, \delta$ is defined analogously to a schedule for tightly-coupled tasks with the additional requirement that for every $t \in T$, $lb(t) < s(t) < ub(t)$ and for every method $m_i$, $s(t_{i,s}) - s(t_{i,s}) = \delta(m_i)$.

A DTN $(T_i)_{i=1}^n, \equiv, I, \delta$ is called globally consistent if there exists at least one schedule $s$ for it. If the DTN is a single agent DTN $(T_i, \equiv, I, \delta)$ it is called locally consistent if there is at least one feasible schedule for it. A DTN $(T_i)_{i=1}^n, \equiv, I, \delta$ is called coordinated if it is globally consistent for every combination of extensions $(M_i, \equiv^+_i, \equiv^+_i)$ of the agent’s DTNs that are locally consistent.

**Reduced DTNs** Remember that plan decoupling for moderately-coupled tasks in fact reduced a coordination problem for moderately-coupled tasks to loosely-coupled tasks, by allowing the agents to plan autonomously.

Before we analyse the decoupling problem for DTNs, in this paragraph, we show that every DTN instance can be reduced to a reduced DTN instance without time windows, obtaining a tightly-coupled task with durations. The idea is to introduce a new time agent $A_0$ that represents an absolute time line, which is a totally-ordered set of time points. Although more time points can be mentioned on that time line, we at least need the upper and lower bounds used in the time windows. Because we defined our time windows as $(lb(t), ub(t))$, precedence constraints can be used to constrain the occurrence of any time point $t$ allowed during execution by $t_{lb(t)} < t < t_{ub(t)}$. 

COPLAS 2007 79
We now give a more formal reduction in which the time window constraints are replaced by a set of precedence constraints using an additional time agent.

Given an instance \( \{(T_i)_{i=1}^n, \prec, \equiv, I, \delta\} \), we first collect all the lower and upper bounds of the time windows \( I(t) \). More precisely, for each time point \( t \), the time window \( I(t) = (lb(t), ub(t)) \) is coded into two values \( s \) and \( s' \), where \( s = t_{lb(t)} \) and \( s' = t_{ub(t)} \). We collect the total set of all these time values in the set \( T_0 = \{s_1, \ldots, s_p\} \) where \( s_1 < \cdots < s_p \). Now, we construct the following coordination instance \( \{(T_i)_{i=0}^n, \prec', \equiv', \delta'\} \), where

1. The partitioned set of tasks is extended to \( \{T_0\} \cup \{T_i\}_{i=1}^n \).
2. The precedence relation \( \prec' \) equals \( \prec \) extended with the total ordering imposed on \( T_0 \). Moreover, for each task \( t \), with associated time window \( I(t) = (s, s') \), two additional precedence constraints \( s \prec t \prec s' \) are constructed. The result then is \( \prec' = \prec \cup \{s \prec t \prec s' \mid t \in \bigcup_{i=1}^n T_i, s = lb(t), s' = ub(t)\} \).
3. The synchronisation relation \( \equiv' \) equals the relation \( \equiv \) extended with the tuples \( t \equiv s_i \), for each time point \( t \) with \( I(t) = (s_i - \varepsilon, s_i + \varepsilon) \) and \( s_i = s_j \).

This can easily be seen by noting that a coordination set only contains precedence intra-agent precedence constraints, the time agent \( A_0 \) is totally ordered, and no time points are added to the set of agents \( A \).

It almost suffices to only use time points with precedence and synchronisation constraints. However, for each method assigned to an agent, we need an ordered pair of time points and its associated duration. Unfortunately, this cannot be represented by using relations among time points alone, and need to label the durative arcs.

As an example, reconsider the moderately-coupled task in Figure 1(a), where all time points are constrained to time window \( (17, 43) \). The possible coordination sets \( \Gamma \) are \( \{t_{1,r} \prec t_{4,s}\}, \{t_{3,r} \prec t_{2,s}\} \), and \( \{t_{1,e} \prec t_{4,s}; t_{3,e} \prec t_{2,e}\} \). For example, if we have \( \delta(m_1) = 19 \), \( \delta(m_2) = 1 \), \( \delta(m_3) = 2 \), and \( \delta(m_4) = 3 \), then \( \{t_{1,e} \prec t_{4,s}\} \) does not coordinate the task, because agent \( A_2 \) can plan \( m_2 \prec m_3 \).

Here, the task is inconsistent and, therefore, not coordinated, because task \( m_4 \) must not be scheduled to start earlier than \( lb(t_{1,s}) + \varepsilon + \delta(m_1) + \varepsilon + \delta(m_2) + \varepsilon + \delta(m_3) + \varepsilon = 17+1+19+1+1+1+2+1 = 43 \) (based on the durations and \( lb(m_1) \)), and be completed before 43. Obviously, there is no schedule in which \( m_4 \) meets these constraints.

The key problem with DTNs is the interaction of time windows and durations of the methods. As we have already seen, it is not possible to discard the method durations when solving the coordination problem. A partial solution to this problem might be to discover all implied constraints first, or otherwise to adapt the problem to prevent these incorrect coordination sets.

Instead of discarding the duration information, we need to use it for temporal constraint propagation and elicitation. This can tighten time windows which can, in turn, result in new precedence constraints because time windows become non overlapping. As an example, we consider the normalisation of a DTN for an agent \( A_1 \), that is assigned methods \( m_2, m_5 \) that are constrained by time windows. In Figure 4, the result is shown after propagating the available temporal information. Here, the constraint \( t_{5,s} \prec t_{2,e} \) emerges through non-overlapping time windows \( (20, 25) \) and \( (28, 65) \).

Note that these constraints are implied without making any assumptions on the degree of parallelism available to that agent. In terms of qualitative temporal constraints, this additional constraint means that \( m_2 \) before \( m_5 \) is excluded. Now, the agent can reason whether other constraints hold due to, for instance, the degree of parallelism available. If an agent is strictly sequential, than it also discards all constraints with (partially) overlapping execution, such that only \( m_5 \) before \( m_2 \) remains. Adding such precedence constraints requires time windows to be tightened again, and possibly resulting in new local precedence constraints.

**Plan coordination with durative tasks** Using DTNs, we can represent methods as (durative) intervals, makes it possible to use qualitative temporal constraints on methods, and allows time windows to constrain the time points to absolute

---

Note: The above text is a continuation of the previous content and has been formatted to maintain coherence and readability. The diagrams and equations have been represented in a textual format due to the limitations of the text-based interface.
time. Note that DTNs have much in common with Simple Temporal Networks (STNs) (Dechter, Meiri, & Pearl 1991). Before continuing, we briefly describe STNs and show that DTNs are in fact restrictions of STNs.

Formally, an STN $S$ is written as a tuple $(T, C)$, where $T$ is the set of time points $\{z, t_1, \ldots, t_m\}$, and $C$ is a finite set of binary constraints on those time points. The point $z$ represents an arbitrary fixed reference point on the time line, commonly referred to as the zero time point. Each constraint $c \in C$ has the form $\delta_{lb} \leq t_j - t_i \leq \delta_{ub}$ for some $\delta_{lb} \in \mathbb{R} \cup \{-\infty\}$, $\delta_{ub} \in \mathbb{R} \cup \{\infty\}$, and commonly is represented as $\delta(t_i, t_j) = [\delta_{lb}, \delta_{ub}]$.

We show that DTNs are in fact restrictions of STNs, by giving a transformation of a DTN $(\{T_i\}_{i=1}^n, \leq, \equiv, I, \delta)$ to its representation as an STN $(T, C)$. First, the time points in an STN consist of the time points from the DTN together with the zero time point $z$. $T = \{z\} \cup \bigcup_{i=1}^n T_i$. Second, it is assumed that all pairs of time points are constrained by the universal temporal constraint $\delta(t, t') = (-\infty, \infty)$, but can be tightened in the following way.

1. For $i, j \forall t \in T_i, t' \in T_j : (t, t') \in \leq$ implies $\delta(t, t') = [1, \infty) \in C$,
2. For $i, j \forall t \in T_i, t' \in T_j : (t, t') \in \equiv$ implies $\delta(t, t') = [0, 0] \in C$,
3. For $i, j \forall t \in T_i : I(t) = (lb(t), ub(t))$ implies $\delta(z, t) = (lb(t), ub(t)) \in C$, and
4. For $m \exists M : \delta(t_s, t_e) = [\delta(m), \delta(m)] \in C$.

In the previous paragraph, we showed that a DTN with time windows can be reduced to a tightly-coupled task with durations, without any new difficulties being introduced. In this paragraph, we analyse the consequences of the added tightly-coupled tasks. Thereafter, we study the complexity of actually determining a coordination set, and look at the issues that need to be taken into account.

**Proposition 1** For any globally consistent DTN $D = (\{T_i\}_{i=1}^n, \leq, \equiv, I, \delta)$, there exists a coordination set $\Gamma = (\Gamma_\leq \cup \Gamma_\equiv)$ such that the resulting DTN is coordinated.

**Proof** Since $D$ is globally consistent, there exists at least one feasible schedule $s$ for it. Define the set $\Gamma_\leq$ as follows: $\forall i, j \forall t', t \in T_i, \forall t' \in T_i : s(t) < s(t')$ iff $(t, t') \in \Gamma_\leq$, and define $\Gamma_\equiv$

\begin{align*}
\Gamma_\equiv & := \{(t, t') \in T_i \times T_j : s(t) = s(t') \text{ iff } (t, t') \in \Gamma_\equiv\}.
\end{align*}

\begin{align*}
\forall i, j \forall t \in T_i, t' \in T_j : (t, t') \in \leq & \text{ implies } \delta(t, t') = [1, \infty) \in C, \\
\forall i, j \forall t \in T_i, t' \in T_j : (t, t') \in \equiv & \text{ implies } \delta(t, t') = [0, 0] \in C, \\
\forall i, j \forall t \in T_i : I(t) = (lb(t), ub(t)) & \text{ implies } \delta(z, t) = (lb(t), ub(t)) \in C, \text{ and}
\forall m \exists M : \delta(t_s, t_e) = [\delta(m), \delta(m)] & \in C.
\end{align*}

In order to minimise the loss of freedom for the individual agents, we again have to identify a smallest set of additional constraints. We call this the PDP for DTNs, for which we define the recognition problem as follows.

**DTN-Coordination Recognition (DTN-CR)**

**INSTANCE:** Globally consistent DTN $D$ partitioned into $n$ agent’s DTNs.

**QUESTION:** Does it hold that the DTN is globally consistent for every set of locally-consistent extensions of individual DTNs $D_i$?

**Lemma 1** DTN-CR is $\text{coNP}$-complete.

**Proof** Membership is shown by noting that a no-certificate is verified in polynomial time. A no-certificate contains a set of precedence and synchronisation constraints that are added to the DTN in polynomial time. As shown above, a DTN can be transformed into an equivalent STN in polynomial time, whose consistency can be checked in polynomial time.

Hardness is proven by a reduction from the PDP-Coordination Recognition problem for tightly-coupled tasks. Here, the idea is to associate time windows with time points that are wide enough such that every partial order is consistent (e.g., set all time windows to $[0, \infty)$). Additionally, we fix all durations to $\delta(m_i) = 1$.

As a corollary, we can say that the more general coordination variant of DTN-CR (i.e., minimally change the DTN such that it is a yes-instance for DTN-CR) is $\Sigma^P_2$-complete. Intuitively, guessing a yes-certificate can be verified in polynomial time using a DTN-CR-oracle, which proves membership. Hardness, on the other hand, can be proven using the same reduction as used in the lemma.

Checking a DTN’s consistency is an important issue in these problem instances. For this paper, STNs are sufficient to express the temporal constraints, because the agent’s degrees of parallelism are not bounded. In the following example, we show a complex task where unions of time windows are needed when this parallelism is bounded. However, consistency checking for such temporal networks is intractable (Dechter, Meiri, & Pearl 1991), and its consequences are part of future research.

Consider the DTN depicted in Figure 5. Here, we have three methods that each have a duration $\delta(m_i) = 3$, methods $m_1, m_2$ need to be scheduled within the time window $(10, 19)$ and $m_3$ in $(0, 30)$. When the agent has unbounded parallelism, no problems occur when adding precedence constraints. However, when the agent is a unit-capacity resource, the temporal network basically becomes a disjunctive temporal network due to $m_3$ becoming constrained to $(0, 11) \cup (18, 30)$.
Figure 5: A union of available time windows can be implied when agent’s parallelism is bounded.

**Relation to Simple Temporal Networks**

In the previous sections, we have taken a plan-decoupling approach to solving the plan-coordination problem. However, instead of decoupling agent’s plans, we could as well decouple agent’s schedules. This approach gives rise to the Temporal-Decoupling Problem (TDP) (Hunsberger 2002), which is defined on STNs.

Basically, an STN is a set of time points and a set of binary constraints between those time points. Commonly, such an STN is represented as a directed graph with numerical values on its arcs. Here, the vertices represent the time points, while the arcs are upper and lower bounds on the temporal distance between two time points.

An STN can be viewed as the set of schedules for the task it represents. Therefore, a solution to an STN is a schedule for completing the methods (i.e., assignment of values to the time point variables such that all constraints are satisfied). An STN is consistent when at least one solution exists.

In Figure 6(a), we have two agents $A_1$, $A_2$ each having one method (i.e., two time points) that takes exactly 4 time units to complete, where $m_1$ needs to be scheduled in $[19, 64]$, and $m_2$ in $[19, 69]$. Moreover, the methods are constrained by $m_1$ before $m_2$, which translates to $t_{1,e} < t_{2,s}$. The constraints between the time points are given as temporal distance intervals $[\Delta_{lb}, \Delta_{ub}]$ on the arcs.

Note that tightening of temporal distances works in a similar way as the tightening of time windows in DTNs. Clearly, in this example, the agents cannot schedule independently due to the precedence constraint $1 \leq t_{2,s} - t_{1,e} < \infty$. In order to allow independent scheduling, we need to temporally decouple the agents. This is achieved when the constraint between $t_{1,e}$ and $t_{2,s}$ is implied by the constraints between $z$ and $t_{1,e}$, and $z$ and $t_{2,s}$, respectively. A typical solution for TDP on the STN of Figure 6(a) is depicted in Figure 6(b). Here, the inter-agent constraint has become implicit, because $23 \leq t_{1,e} - z \leq 48 \leq 49 \leq t_{2,s} - z \leq 65$, results in $1 = 49 - 48 \leq t_{2,s} - t_{1,e} \leq 65 - 23 = 42 < \infty$.

Basically, the difference between PDP and TDP is the following. In TDP, the problem is to tighten the time windows of the time points in such a way that the inter-agent temporal differences are guaranteed to hold (e.g., by making them implicit) and the joint schedule to be feasible, whatever schedules is constructed by the individual agents. In PDP, the problem is to reduce the local planning freedom such that the joint plan is guaranteed to be feasible and to be consistent, whatever plans is constructed by the individual agents.

Clearly, there is a close resemblance between STNs and DTNs. For instance, both frameworks use time points and binary constraints between them as basic constructs, and time windows on time points in DTNs are represented as constraints between $z$ and that point in STNs. The major difference between the two representations is that in STNs every pair of time points can be constrained by a temporal distance interval, where in DTNs only fixed temporal distances are allowed between time points within an agent.

Although the frameworks in which these problems are studied show much likeness, the problems themselves largely differ. Even the basic approach, as in a decoupling, is very similar, although one plan-decouples the agents by guaranteeing the existence of at least one schedule, while in temporal-decoupling each agent is left with its decoupled part of the STN which represents the set of schedules. From a complexity point of view, there is a big difference, because PDP is $\Sigma^P_2$-complete while solving a TDP is in P. In fact, we believe that TDP over-constrains complex tasks in order to coordinate them, in the sense that it reduces the number of possible plans more than needed.

**Conclusions and Future Work**

Plan coordination is needed to guarantee that using local planning autonomy does not cause conflicts to the global goal. Here, a distinction can be made between pre-, interleaved, and post-planning coordination. Both interleaved and post-planning coordination assume communication to be available during and after planning and thus during execution. In many domains, however, communication can be lost or difficult to establish or maintain, or agents are unwilling to revise their plans. Therefore, interleaved and post-planning coordination are not always applicable, and we chose to take a pre-planning approach to coordination.

In this paper, we presented a framework for modelling complex tasks. Using precedence and synchronisation constraints to constrain the relative execution of time points, all qualitative temporal constraints can be used to constrain two methods. This framework was further extended to allow methods to have a certain fixed duration, and to be executed within a certain time window. We discovered that plan coordination can be achieved when reasoning with qualitative temporal constraints and durative tasks, and defined and
studied the complexity of this new coordination problem.

With respect to the use of temporal constraints, existing work on plan coordination has been rather limited. Although the described temporal-decoupling problem is a good start at coordinating problems with quantitative temporal information, its use is rather limited. First, it is a schedule-Coordination approach that reduces the planning freedom to a greater extend than necessary. Second, the used STNs are a small subset of instances that can be defined in temporal constraint networks as used in general temporal networks (Dechter, Meiri, & Pearl 1991), where it is possible to represent arbitrary intervals of temporal distances between time points. In the future, it would be interesting to combine the pre-planning and pre-scheduling coordination approaches using an even more expressive framework including both quantitative and qualitative temporal information.

In future research, some additional steps are needed to make our approach applicable to real-life problems in real-time. First, in many domains, uncertainties (e.g., on duration) need to be taken into account, for which an extension of STNs has been developed, called STNUs (Vidal & Fargier 1999). Second, until now, we have only been concerned with static problem instances. Reality, however, is much more dynamic than these situations, and a technique is needed to deal with this additional dynamism online. In (Hunsberger 2003), STNs are adapted to Augmented STNs(ASTNs) to cope with the passing of time, in which new methods and constraints can be inserted into an existing ASTN.

Finally, the plan-coordination problems should be formulated as optimisation problems, because this corresponds more closely to reality. In order to do this, we need to look at different criteria for the optimal coordination set. It is not hard to come up with examples where one coordination set tightens time windows, while another coordination set does not tighten any time window. Clearly, the latter solution reduces the scheduling freedom less than the first, in an absolute sense, and is likely to be preferred.

Acknowledgements

J. Renze Steenhuisen is supported by the Dutch Ministry of Economic Affairs, grant nr: BSIK03024. The ICIS project is hosted by the DECIS Lab, the open research partnership of Thales Nederland, the Delft University of Technology, the University of Amsterdam, and the Netherlands Foundation of Applied Scientific Research (TNO).

References


Constraint-Based Modelling of Discrete Event Dynamic Systems

Gérard Verfaillie and Cédric Pralet and Michel Lemaître
ONERA, 2 av. Édouard Belin, BP 74025, F-31055 Toulouse Cédex 4, France

Abstract
Numerous frameworks dedicated to the modelling of discrete event dynamic systems have been proposed to deal with programming, simulation, validation, situation tracking, or decision tasks (automata, Petri nets, Markov chains, temporal logic, situation calculus, STRIPS . . .). All these frameworks present significant similarities, but none offers the flexibility of more generic frameworks such as logics or constraints.

In this article, we propose a generic framework for the modelling of discrete event dynamic systems, whose main components are state and event timelines and constraints on these timelines. Although any kind of constraint can be defined on timelines, we focus on some useful ones: pure temporal constraints, instantaneous state and event constraints, instantaneous and non instantaneous transition constraints.

Finally we show how the proposed framework subsumes existing apparently different frameworks such as automata, Petri nets, or classical frameworks used in planning and scheduling, while offering the great flexibility of a constraint-based modelling.

Introduction
The goal of this article is to propose a generic constraint-based framework for the modelling of discrete event dynamic systems, that is of systems whose state evolves over time via instantaneous changes possibly due to instantaneous events.

Numerous frameworks exist to model such systems. One can cite automata, synchronous languages (Benveniste et al. 2003) which allow automata to be compactly described, and temporal logics (Pnueli 1977) which allow properties of automata to be compactly described, but also Petri nets, Markov chains and Markov Decision Processes (Puterman 1994) which both allow stochastic changes to be described, the STRIPS framework (Fikes & Nilsson 1971) and the situation calculus (Levesque et al. 1997) both used in planning, and the usual models used in scheduling.

Although all these frameworks present significant similarities (discrete instants of transition, more or less compact representation of states and transitions), comparing them is somewhat difficult, unless translating all of them into the most basic ones and less compact ones: automata or Markov chains.

On the other hand, although constraint-based modelling is known to combine compactness and flexibility in terms of modelling with efficiency in terms of problem solving, it remains mainly used to deal with static problems, that is problems that do not involve time, despite some notable exceptions: mainly the scheduling problems (see (Baptiste, Pape, & Nuijten 2001)) and to a certain extent planning problems (see for example (Kautz & Selman 1992; van Beek & Chen 1999)). With only a few exceptions (see for example (Delzanno & Podelski 2001)), it is not used to deal with validation problems on dynamic systems or with situation tracking problems, such as failure diagnosis. We think that such a situation is mainly due to the absence of a generic constraint-based framework, dedicated to the modelling of discrete event dynamic systems and indifferently usable for simulation, validation, situation tracking, or decision tasks.

This is such a framework we propose in this article. It is based on the assumption of a continuous time and of discrete instants of event or change, and on the notion of timelines: state timelines to represent the way the state of the system evolves over time and event timelines to represent the way events occur. These timelines can be compactly represented via variables: temporal variables to represent the instants of change or event, and atemporal variables to represent values at these instants. Using the great flexibility of a constraint-based modelling, any kind of constraint can be defined on these timelines via constraints on temporal and atemporal variables. However, among them, pure temporal, instantaneous state, instantaneous event, instantaneous transition, and non instantaneous transition constraints are a priori very useful and would deserve to be particularly studied.

The framework proposed in this article is inspired but different from works carried out at the frontier between planning and scheduling problems, where the notion of timeline is used to represent the way state and resources evolve over time and to reason on time, state, and resources (Lahorée & Ghalbab 1995; Muscettola 1994; Ghalbab 1996; Muscettola et al. 1998; Barták 1999; Frank & Jönsson 2003).

Section Modelling Assumptions introduces basic assumptions related to time, states, and events. Section Time-lines introduces the timeline-based representation, whereas Section Constraint networks on timelines defines what is a constraint network on timelines (CNT) and Section Use-
ful types of constraint focuses on some \textit{a priori} very useful types of constraint. In section \textbf{Subsumed frameworks}, we show how the proposed framework subsumes automata, Petri nets and classical frameworks used in planning and scheduling. Section \textbf{What remains to be done} concludes with the remaining work and some possible extensions.

This article focuses on \textit{modelling} issues and says nothing about \textit{algorithmic} issues (constraint propagation, search, . . .), which will be the subject of future studies and articles. We do that because we think that the first obstacle, and perhaps the main one, to the systematic use of constraint-based modelling and reasoning in the context of discrete event dynamic systems is the modelling question.

Note that this work has nothing to do with the works on \textit{dynamic CSPs} (Verfaillie & Jussien 2005). Dynamic CSPs aim at dealing with dynamic models, that is with changes which may occur in CSP models. In this work, we want to deal with static models of dynamic systems, that is with static models which include the system dynamics.

\section*{Modelling Assumptions}

\subsection*{Time}

We want to reason on instants, on the \textit{order} between them, but also on their \textit{values}. These values are assumed to belong to a \textit{continuous} set. This is why we use $\mathbb{R}$, with the natural order over reals, to model time.

\subsection*{States and State Changes}

\textbf{States} We assume that the state of a system can be modelled using a finite set of \textit{state variables} representing the attributes of the state of this system. With each state variable, is associated a \textit{domain} of values which can be finite or infinite, continuous or discrete, symbolic or numeric. In such conditions, the state of the system at any time is modelled by an assignment to each state variable of a value in its domain.

It must be noted that state variables can be used to represent passive attributes of the state (such as, for a robot, its position or its available level of energy), as well as active ones (such as, still for a robot, the mode of an observation instrument or the fact that the robot is currently moving in some way). In other words, state variables can be used to represent what we usually refer to as the state of the system (position, energy level, . . .), as well as what we usually refer to as \textit{actions}, when they are not instantaneous (an observation, a movement, . . .).

\textbf{State Changes} We assume that the state of a system can change via \textit{instantaneous changes} and only this way. Continuous changes cannot be hence precisely modelled and only approximated via a sequence of instantaneous changes. In such conditions, a change in the state of the system is modelled by an instantaneous simultaneous change in the assignment of a non empty subset of the state variables. Moreover, we adopt the convention that, when the assignment of a state variable $v$ changes at time $t$ from value $val$ to value $val'$, $v$ is assigned value $val$ before $t$, $t$ excluded, and value $val'$ after $t$, $t$ included\footnote{This convention, used for example in \textit{synchronous languages}, is very useful to model instantaneous events which lead to instantaneous changes at the same time, for example a failure which leads instantaneously a system to a given failure mode.}

State changes can occur at any time, but we assume that the instants at which they occur form a \textit{discrete} subset of $\mathbb{R}$. Consequently, the assignment of a state variable remains constant from an instant $t$ of change to the next instant $t'$ of change, that is equal to the value it took at $t$ over the semi-closed interval $[t, t']$ (see Figure 1).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{state_variable_over_time.png}
\caption{State variable over time}
\end{figure}

\subsection*{Events and Event Occurrences}

\textbf{Events} The same way we assumed that the state of a system can be modelled using a finite set of state variables, we assume that the events that may occur can be modelled using a finite set of \textit{event variables}. With each event variable, is associated a \textit{domain} of values which can be finite or infinite, continuous or discrete, symbolic or numeric. At this domain, we systematically add a value \textit{nothing} ($\bot$) to represent the absence of value. In such conditions, at any time, the set of events that are present is modelled by an assignment to each event variable of a value in its domain, possibly equal to $\bot$.

It is for example possible to associate an event variable with each type of event, with its value pointing out its content and the value $\bot$ pointing out the absence of event of this type.

\textbf{Event Occurrences} We assume that events are \textit{instantaneous phenomena}. In such conditions, an event occurrence is modelled by an instantaneous simultaneous assignment of a value different from $\bot$ to a non empty subset of the event variables.

Events can occur at any time but, as with state changes, we assume that the instants at which they occur form a \textit{discrete} subset of $\mathbb{R}$. Consequently, the assignment of an event variable remains equal to $\bot$ between two successive instants of event $t$ and $t'$, $t$ and $t'$ excluded, that is on the open interval $(t, t')$ (see Figure 2).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{event_occurrence.png}
\caption{Event occurrence}
\end{figure}

\subsection*{State Changes and Event Occurrences}

No assumption is \textit{a priori} made about any correlation or causality relation between state changes and event occurrences. State changes and event occurrences can be simultaneous. State changes can occur without any event and events without any state change.
Timelines

In this section, we show how timelines can be used to represent compactly the way state and event variables evolve over time.

Definition 1 A timeline \( tI \) is defined as a quintuple \( \langle v, d, I, tI, tvI \rangle \) where \( v \) is a state or event variable, \( d \) its domain of values, \( I \) a sequence of instants, \( tI \) a sequence of temporal variables, and \( tvI \) a sequence of atemporal variables.

If \( v \) is a state variable, then we speak of a state timeline. Else, we speak of an event timeline.

Sequence \( I \) is assumed to be countable\(^2\). Thus, \( I \) can be seen as a sequence of instant indices. Let be \( I = [1, \ldots, i, \ldots, l] \), \( I^+ = [0, 1, \ldots, i, \ldots, l] \), and \( I^- = [2, \ldots, i, \ldots, l] \). Sequence \( tI \) associates with each instant \( i \in I \) a temporal variable \( tI_i \in \mathbb{R} \) which represents the temporal position of instant \( i \). Sequence \( tvI \) associates with each instant \( i \in I^+ \) an atemporal variable \( tvI_i \in d \) which represents the value of \( v \) at instant \( i \).

Instants are temporally ordered. So, we enforce that \( \forall i \in I^- \), \( tI_{i-1} \leq tI_i \) and (\( tI_i = tI_{i-1} \)) \( \rightarrow \) (\( tvI_i = tvI_{i-1} \)). Moreover, in case of an event timeline, we enforce that \( tvI_0 = \perp \).

A timeline \( tI = \langle v, d, I, tI, tvI \rangle \) represents the way \( v \) evolves over time. Sequence \( I \) is the sequence of the instants at which changes or events may occur (they may occur, but are not mandatory), \( tI \) is the sequence of their temporal positions, and \( tvI \) the sequence of values of \( v \) at these instants. The first instant (\( 0 \)) in this sequence is fictitious and has no associated temporal position. It is used to represent the initial value of \( v \), equal to \( \perp \) in case of an event variable (no event at the initial instant). Figure 3 shows the tabular representation of such a timeline.

| \( t \) | \( 0 \) | \( 1 \) | \( \ldots \) | \( i \) | \( \ldots \) |
| \( v \) | \( v_0 \) | \( v_1 \) | \( \ldots \) | \( v_i \) | \( \ldots \) |

Figure 3: Tabular representation of a timeline

It is important not to mistake the sequence \( I \) of instants for the sequence \( tI \) of their temporal positions. From now, it is also important not to mistake state and event variables for temporal and atemporal variables that appear in timelines. Only the latter are actually mathematical variables.

The former are functions over time. When confusion will be possible, we will keep the term variable for temporal and atemporal variables and use the term timeline for state and event variables. Moreover, when no confusion will be possible, we will speak indifferently of \( v \) and \( tI \), making no distinction between a state or event variable and its associated timeline.

A timeline \( tI = \langle v, d, I, tI, tvI \rangle \) is said to be finite if \( I \) is finite. It is said to be completely assigned if all the temporal and atemporal variables in \( tI \) and \( tvI \) are assigned. Let \( tI = \langle v, d, I, tI, tvI \rangle \) be a finite completely assigned timeline, with \( I = [1, \ldots, i, \ldots, l] \). We refer to \( l \) as its length and to the closed interval \([tI_i, tI_l] \) as its temporal horizon \( H \).

From the assumptions of Section Modelling Assumptions (a state variable remains constant and an event variable remains equal to \( \perp \) between two successive instants of change or event), it is easy to derive from any finite completely assigned timeline \( tI = \langle v, d, I, tI, tvI \rangle \) the function which associates with any \( t \in H \) (and not only with any \( t \in I \)) the value that \( v \) takes at \( t \) (when \( i = \max \{ i' \mid I \subseteq I \} \) for a state timeline; \( v \) if there exists \( i \in I \) such that \( tI_i = tI_{i-1} \) and \( \perp \) otherwise for an event timeline) but also the value it takes just before \( tI_i (v_{i-1}\) for a state timeline; \( \perp \) for an event timeline) and the one it takes just after \( tI_i (v_{i+1}\) for a state timeline; \( \perp \) for an event timeline). Figure 4 shows a partial graphical representation of this function: piecewise constant function for a state timeline and multi-dirac function for an event one.

![Figure 4: Functions over time, associated with a finite completely assigned timeline of length \( l \), in case of a state timeline (above) or an event timeline (below)](image)

Constraint networks on timelines

Constraint network definition

In this section, we show how constraints can be defined on timelines, in order to represent the combined evolutions of the state and event variables that are either possible or required.

Definition 2 A constraint network \( CNT \) on timelines is a pair \( (TL, C) \) where:

- \( TL \) is a finite set of timelines which all share the same sequence \( I \) of instants and the same sequence \( tI \) of their temporal positions;

\(^2\)A set is denumerable if and only if it is equipollent to the finite ordinals. It is countable if and only if it is either finite or denumerable.
• C is a finite set of constraints on the timelines in TL (see definition\(^3\)).

We note \( V = \{v | \langle v, d, I, t_i, v_i \rangle \in TL \} \), \( \forall i \in I^+ \), \( V_i = \{v_i | v \in V \} \), \( V_I = \{v_i | i \in I^+ \} \). SV, SV\(_I\), and SV\(_T\) (respectively EV, EV\(_I\), and EV\(_T\)) can be similarly defined by restricting ourselves to state timelines (respectively event timelines). Finally, we note \( V_{ar} = t_I \cup V_I \)

\( t_I \) is the set of temporal variables in the CNT, \( V_I \) the set of atemporal variables, and \( V_{ar} \) the whole set of variables, either temporal or atemporal. The same way as with timelines, we can define what a finite CNT and a completely assigned one are.

**Definition 3** A constraint \( c \) on a set TL of timelines is a quadruple \( \langle qt, cd, sc, df \rangle \) where:

• \( qt \) is a finite sequence \( [q_1, \ldots, q_m] \) of quantifiers, with \( q_j \in \{\forall, \exists\} \);

• \( cd \) is a finite sequence \( [c_1, \ldots, c_m] \) of conditions, each condition \( c_j \) being a boolean function over \( I^+ \);

• \( sc \) is a function which associates with any sequence \( [i_1, \ldots, i_m] \in I^m \) satisfying the conditions in \( cd \) a basic constraint scope \( sc(i_1, \ldots, i_m) \), that is a finite sequence of variables in \( V_{ar} \);

• \( df \) is a function which associates with any sequence \( [i_1, \ldots, i_m] \in I^m \) satisfying the conditions in \( cd \) a basic constraint definition \( df(i_1, \ldots, i_m) \), that is a boolean function over the Cartesian product of the domains of the variables in \( sc(i_1, \ldots, i_m) \).

If \( m = 0 \), then \( qt = \emptyset \) and \( qt \neq \emptyset \), \( sc \) is a basic constraint scope, and \( df \) a basic constraint definition.

A basic constraint is a classical CSP constraint, defined as usual by its scope \( sc \), which is a finite sequence of variables, and its definition \( df \), which is a boolean function over the Cartesian product of the domains of the variables in \( sc \) (Rossi, Beek, & Walsh 2006). Quantification \( qt \) is used to specify in one non basic constraint a possibly infinite set of basic constraints by iterating on \( I \) which may be infinite. Condition \( cd \) is used to limit the elements of \( I \) on which to iterate. Functions \( sc \) and \( df \) are used to associate a basic constraint, that is a scope \( sc(i_1, \ldots, i_m) \) and a definition \( df(i_1, \ldots, i_m) \), with any sequence \( [i_1, \ldots, i_m] \in I^m \). Scopes can be specified by extension when \( I \) is finite or \( m = 0 \). They must be specified by intension otherwise. Definitions can be specified by extension when \( I \) is finite or \( m = 0 \) and when the domains of the involved variables are finite. They must be specified by intension otherwise.

To take a very simple example, let us consider a system which is represented by one state variable \( v \) whose value changes at each instant. We want to express that \( \forall i \in I, v_i \neq v_{i-1} \). The associated CNT constraint is \( c = \langle qt, cd, sc, df \rangle \) where \( qt = [\forall] \) (sequence of quantifiers reduced to the only quantifier \( \forall \)), \( cd = [\text{true}] \) (no condition on \( I \)), and \( \forall i \in I, sc(i) = [v_i, v_{i-1}] \) (scope limited to variables \( v_i \) and \( v_{i-1} \)) and \( df(i) \equiv (v_i \neq v_{i-1}) \) (definition given by the \( \neq \) relation between both variables).

In spite of the presence of quantifiers, it is important not to mistake this framework with the Quantified CSP framework (Borner et al. 2003). Here, quantification is associated with variable indices and used to specify compactly possibly infinite sets of constraints, whereas quantification is associated with variable values in the QCSP framework.

**Constraint satisfaction**

Let us consider a finite CNT \( \langle TL, C \rangle \) and a complete assignment \( A \) of it, that is of the set \( V_{ar} \) of involved variables. We can define recursively what is the satisfaction of a constraint \( c \in C \) by \( A \).

**Definition 4** A complete assignment \( A \) of a finite CNT \( \langle TL, C \rangle \) satisfies a constraint \( c \in C, c = \langle qt, sc, df \rangle \) if and only if it satisfies the quadruple \( \langle \emptyset, qt, sc, df \rangle \). A complete assignment \( A \) of a finite CNT satisfies a quadruple \( \langle Is, qt, sc, df \rangle \), where \( Is \) is a sequence of elements of \( I \), if and only if:

• if \( qt = \emptyset \): \( (df(Is))(A_{sc(Is)}) = true \)

• if \( qt = [q] \cup qt' \):

  - if \( q = \forall \): \( \forall i \in I \) such that \( cd(Is \cup [i]) \), \( A \) satisfies \( \langle Is \cup [i], qt', sc, df \rangle \);

  - if \( q = \exists \): \( \exists i \in I \) such that \( cd(Is \cup [i]) \) and \( A \) satisfies \( \langle Is \cup [i], qt', sc, df \rangle \).

In the first case (empty sequence of quantifiers), the quadruple specifies a basic CSP constraint and constraint satisfaction is defined as usual in the CSP framework. The second case (non empty sequence of quantifiers), can be split into two sub-cases according to the first quantifier in the sequence: \( \forall \) or \( \exists \). Note that a universal quantifier leads to a conjunction of constraints, whereas an existential one leads to a disjunction.

We say that a complete assignment \( A \) of the variables \( V_{ar} \) of a finite CNT is consistent if and only if it satisfies all the constraints in \( C \).

**Complexity of constraint checking**

If all the variables have finite domains of values of maximal size \( md \), if all the basic constraints implicitly defined by the non basic ones are of maximal arity \( ma \), if all the non basic constraints have sequences of quantifiers of maximal size \( ms \), and if the CNT is of maximal length \( l \), then the time complexity of checking the satisfaction of a constraint by a complete assignment is \( O(l^{ms} \cdot c(ma, md)) \), if we note \( c(ma, md) \) the time complexity of checking the satisfaction of a basic constraint of maximal arity \( ma \) over domains of maximal size \( md \). Without any surprise, this complexity grows exponentially with the maximal size \( ms \) of the sequences of quantifiers used in the constraint specifications.

**Useful types of constraint**

Section Constraint network definition introduced a very generic way of specifying constraints on timelines. But, it may be interesting to focus on some specific cases which may be a priori very useful when modelling and reasoning on discrete event dynamic systems. In this section, we
consider pure temporal, instantaneous state, instantaneous event, instantaneous transition, and non instantaneous transition constraints.

Pure temporal constraints
Pure temporal constraints, which involve only temporal variables, are useful to constrain the temporal positions of the instants in the timelines.

A pure temporal constraint is defined as a constraint where scopes \( sc(i_1, \ldots, i_m) \) are made only of variables in \( t_i: \forall [t_1, \ldots, t_m] \in I^m, sc(i_1, \ldots, i_m) \leq t_i \).

A stronger interesting restriction would consist in limiting to 2 the arity of the basic constraints and in enforcing that their definitions be of the form \( df(i, 1_2) \equiv ((t_{i_0} - t_{i_0}) \in [lb, ub]) \) in case of binary constraints and \( df(1_i) \equiv (t_1 \in [lb, ub]) \) in case of unary constraints, resulting in only simple temporal constraints (Dechter, Meiry, & Pearl 1991).

Note the presence of implicit simple temporal constraints in each timeline, enforcing that \( \forall i \in I - t_{i-1} \leq t_i \). These constraints can be modelled using one non basic constraint \( c = (qt, cd, sc, df) \), where \( qt = [\forall] \), \( cd = [i > 1] \), and \( \forall i \in I - t_{i-1}, sc(i) = [t_{i-1}, t_i] \) and \( df(i) = (t_{i-1} \leq t_i) \).

Instantaneous state constraints
Instantaneous state constraints involve only atemporal state variables at the same instant, at which can be added the temporal variable at this instant. They are useful to express the combinations of values of the state variables at the same instant that are possible or required, possibly depending on the temporal position of this instant.

An instantaneous state constraint is defined as a constraint where the length of the sequence of quantifiers is limited to 1 and \( \forall i \in I, sc(i) \subseteq SV \cup \{t_i\} \).

For example, let us assume a robot equipped with two observation instruments which cannot be simultaneously active. This requirement can be modelled using two state timelines is1 and is2, each one with a boolean domain, representing the activity state of each instrument, with \( true \) associated with instrument activity, and one non basic constraint \( c = (qt, cd, sc, df) \), where \( qt = [\forall] \), \( cd = [true] \), and \( \forall i \in I, sc(i) = is1 \cup is2 \) and \( df(i) = (is1 \land is2) \). This constraint specifies that \( \forall i \in I, (e \land is1 \land is2) \).

To take another example, let us assume that we want the robot to be at a given location loq by time \( t_G \). This requirement can be modelled using one state timeline lo, representing the robot location, and one non basic instantaneous state constraint \( c = (qt, cd, sc, df) \), where \( qt = [\exists] \), \( cd = [true] \), and \( \forall i \in I, sc(i) = [lo, t_i] \) and \( df(i) = (lo_i = lo_G) \land (t_i \leq t_G) \). This constraint specifies that \( \exists i \in I \) such that \( (lo_i = lo_G) \land (t_i \leq t_G) \).

Instantaneous event constraints
Instantaneous event constraints involve only atemporal event variables at the same instant, at which can be added the temporal variable at this instant and atemporal state variables at the previous instant. They are useful to express the combinations of values of the event variables at the same instant that are possible or required, possibly depending on the temporal position of this instant and on the combinations of values of the state variables just before this instant, in order to model for example action preconditions.

An instantaneous event constraint is defined as a constraint where the length of the sequence of quantifiers is limited to 1 and \( \forall i \in I, sc(i) \subseteq EV \cup \{t_i\} \cup SV - 1 \).

For example, let us assume a robot which has at its disposal a finite set \( A \) of actions, which cannot be simultaneously triggered. Moreover, let us assume that each action \( a \in A \) requires a level \( e(a) \) of energy to be triggered. This requirement can be modelled using one event timeline ce representing the triggered action, with a domain equal to \( A \cup \{\perp\} \) (\( \perp \) if no action is triggered), state timeline \( sc(i) = [ca_i, ce_{i-1}] \) and \( df(i) = (ca_i \neq \perp \rightarrow (ce_{i-1} \geq e(ca_i))) \). This constraint specifies that \( \forall i \in I, ((ca_i \neq \perp) \rightarrow (ce_{i-1} \geq e(ca_i))) \).

Instantaneous transition constraints
Instantaneous transition constraints involve only atemporal state or event variables at the same instant, at which can be added the temporal variable at this instant and atemporal state variables at the previous instant. They are useful to express the combinations of values of the state and event variables at the same instant that are possible or required, possibly depending on the temporal position of this instant and on the combinations of values of the state variables just before this instant, in order to model for example instantaneous effects.

An instantaneous transition constraint is defined as a constraint where the length of the sequence of quantifiers is limited to 1 and \( \forall i \in I, sc(i) \subseteq SV \cup \{t_i\} \cup SV - 1 \).

For example, let us consider an impulse switch whose position (open or close) can change in case of any impulse. However, let us assume that this switch may fail by remaining stuck at the position it had before failure. These facts can be modelled using three timelines, each one with a boolean domain, and one non basic instantaneous transition constraint. A first state timeline \( sp \) represents the current switch position (open or close), with \( true \) associated with open. A second state timeline \( st \) represents the state of the switch (stuck or not), with \( true \) associated with stuck. A third event timeline \( im \) represents the current impulse (present or not), with \( true \) associated with present and \( false \) with absent \( (\perp = false) \). The physical constraints are represented by one non basic instantaneous transition constraint \( c = (qt, cd, sc, df) \), where \( qt = [\forall] \), \( cd = [true] \), and \( \forall i \in I, sc(i) = [sp_i, sp_{i-1}, st_i, im_i] \) and \( df(i) = (sp_i = sp_{i-1} \leftrightarrow (st_i \land im_i)) \) expressing that the switch position can change if and only the switch is not stuck and an impulse occurs. This constraint specifies that \( \forall i \in I, ((sp_i \neq sp_{i-1}) \leftrightarrow (st_i \land im_i)) \).

Non instantaneous transition constraints
Non instantaneous transition constraints are a bit more complex. They involve atemporal state or event variables between two instants \( i_1 \) and \( i_2 \), including, at which can be added the temporal variables at instants \( i_1 \) and \( i_2 \), and
atemporal state variables at instant $i_1 - 1$. They are useful to express the combinations of values of the state and event variables that are possible or required between two instants, possibly depending on the temporal position of both instants and on the combinations of values of the state variables just before the first instant, in order to model for example non instantaneous action effects.

A non instantaneous transition constraint is defined as a constraint where the length of the sequence of quantifiers is limited to 2 and $\forall\{i_1, i_2\} \in I^2$ such that $i_1 < i_2$, $sc(i_1, i_2) \subseteq \bigcup_{t_1 \leq i < i_2} V_t \cup \{t_1\} \cup \{t_2\} \cup SUV_{i_2-1}$.

For example, let us assume a robot which has at its disposal a finite set $A$ of actions, which cannot run simultaneously for any reason. Moreover, let us assume that each action $a \in A$ has a duration which is not precisely known, but belongs to an interval $[d_{min}(a), d_{max}(a)]$, and that an action $a$ cannot be immediately followed by the same action $a$. These facts can be modelled using one state timeline $ca$ representing the current action, with a domain equal to $A$, at which we can add a special value representing the absence of current action, and one non basic non instantaneous transition constraint $c = \{qt, cd, sc, df\}$, where $qt = [\forall, \exists, \langle\rangle$, $cd = [true, i_1 < i_2]$, and $\forall\{i_1, i_2\} \in I^2$, $sc(i_1, i_2) = [\{ca_{i_1-1}, ca_{i_1} = a\} \cap \{ca_{i_2} \neq a\} \cap (d_{min}(a) \leq (t_2 - t_1) \leq d_{max}(a))]$.

If actions in $A$ produce some resource, such as on-board memory in case of data downloading, and that production is effective at the end of each action, each action $a \in A$ producing $r(a)$, this can be modelled by using a state timeline $cr$ representing the current level of this resource and by adding $cr_{i_2-1}$ and $cr_{i_2}$ in scopes $sc(i_1, i_2)$, and condition ($cr_{i_2} = cr_{i_2-1} + r(a)$) in the right side of definitions $df(i_1, i_2)$.

**Subsumed problems**

In this section, we show how the proposed framework subsumes existing ones such as automata, Petri nets, STRIPS planning, as well as classical models used in scheduling.

**Automata**

An automaton is usually defined as a quadruple $\langle S, E, T, s_0 \rangle$ where:

1. $S$ is a finite set of states;
2. $E$ is a finite set of transition labels;
3. $T \subseteq S \times E \times S$ is a set of transitions;
4. $s_0 \in S$ is the initial state.

An automaton specifies possible transitions; a transition $e \in E$ is possible from state $s \in S$ to state $s' \in S$ if and only if $(s, e, s') \in T$. It is easy to show that an automaton $\langle S, E, T, s_0 \rangle$ is equivalent to a CNT $\langle TL, C \rangle$ where:

1. $TL$ is made of two timelines: one state timeline $cs$ of domain $S$ and one event timeline $e$ of domain $E \cup \{\bot\}$;
2. $C$ is made of two constraints:
   (a) an instantaneous state basic constraint $c_0 = \langle qt_0, cd_0, sc_0, df_0 \rangle$, with $qt_0 = \emptyset$, $cd_0 = \emptyset$, $sc_0 = \{cs_0\}$, and $df_0 \equiv (cs_0 = s_0)$, specifies the initial state;
   (b) an instantaneous transition constraint $c = \langle qt, cd, sc, df \rangle$, with $qt = [\forall]$, $cd = [true]$, and $\forall i \in I, sc(i) = [cs_{i-1}, e_i, cs_i]$ and $df(i) \equiv ((cs_{i-1}, e_i, cs_i) \in T)$, specifies the following possible transitions.

Constraints on CNT appear as a compact way of specifying synchronized products of automata [Arnold & Nivat 1982].

**Petri nets**

A Petri net is usually defined as a quadruple $\langle P, T, Ip, Op \rangle$ where:

1. $P$ is a finite set of places;
2. $T$ is a finite set of transitions;
3. $Ip$ is an input function from $P \times T$ to $N$, which associates a positive integer (possibly null) with each place $p \in P$ and each transition $t \in T$;
4. $Op$ is a similar output function.

A marking $m$ (which can be considered as a state) is defined as a function from $P$ to $N$, which associates an integer $m(p)$ with each place $p \in P$. To be triggered from marking $m$, a transition $t \in T$ must satisfy the following condition: $\forall p \in P, m(p) \geq Ip(p, t)$. If a transition $t \in T$ is triggered from marking $m$, the result is a marking $m'$ where $\forall p \in P, m'(p) = m(p) - Ip(p, t) + Op(p, t)$. As with automata, it is easy to show that a Petri net $\langle P, T, Ip, Op \rangle$ is equivalent to a CNT $\langle TL, C \rangle$ where:

1. a state timeline $mp$ of domain $N$ is associated with each place $p \in P$, at which we add one event timeline $e$ of domain $T \cup \{\bot\}^4$;
2. $C$ is made of two sets of constraints:
   (a) an instantaneous event constraint $c_{p,t}^E = \langle qt_{p,t}^E, cd_{p,t}^E, sc_{p,t}^E, df_{p,t}^E \rangle$ is associated with each place $p \in P$ and each transition $t \in T$, with $qt_{p,t}^E = [\forall]$, $cd_{p,t}^E = [true]$, and $\forall i \in I, sc_{p,t}^E(i) = [e_i, mp_{p,i-1}]$ and $df_{p,t}^E(i) \equiv ((e_i = t) \rightarrow (mp_{p,i-1} \geq Ip(p, t)))$, to specify transition preconditions;
   (b) an instantaneous transition constraint $c_{p,t}^T = \langle qt_{p,t}^T, cd_{p,t}^T, sc_{p,t}^T, df_{p,t}^T \rangle$ is associated with each place $p \in P$ and each transition $t \in T$, with $qt_{p,t}^T = [\forall]$, $cd_{p,t}^T = [true]$, and $\forall i \in I, sc_{p,t}^T(i) = [e_i, mp_{p,i-1}, mp_{p,i}]$ and $df_{p,t}^T(i) \equiv ((e_i = t) \rightarrow (mp_{p,i} = mp_{p,i-1} - Ip(p, t) + Op(p, t)))$, to specify transition effects.

---

4 In a Petri net, two transitions cannot be triggered at the same time.
STRIPS planning

Planning problems may be of very different kind and, despite many efforts, there is no unique framework able to cover all of them [Ghallab, Nau, & Traverso 2004]. This is why we restrict ourselves to the most classical one: the STRIPS framework [Fikes & Nilsson 1971] where a planning problem is defined as a quadruple \((F, A, Is, G)\) where:

1. \(F\) is a finite set of boolean variables, called fluents;
2. \(A\) is a finite set of actions, where each action \(a \in A\) is defined by a triple \((p_a, e_a^+, e_a^-)\), with \(p_a, e_a^+, e_a^- \subseteq F\) and \(e_a^+ \cap e_a^- = \emptyset\), where \(p_a\), \(e_a^+\), and \(e_a^-\) are action preconditions, negative effects, and positive effects;
3. \(Is \subseteq F\) is a set of fluents, which defines the initial state;
4. \(G\) is a finite set of logical conditions on \(F\), called goals.

A state \(s\) is defined as a function from \(F\) to \(B\), which associates a boolean value \(s(f)\) with each fluent \(f \in F\). The following conditions must be satisfied by states and transitions:

1. in the initial state \(s_0\), \((s_0)(f)\) if and only if \(f \in Is\);
2. an action \(a \in A\) can be executed in a state \(s\) if and only if \(\forall f \in p_a, s(f)\);
3. if an action \(a \in A\) is executed in a state \(s\), the result is a state \(s'\) where:
   (a) \(\forall f \in e_a^-\), \(\neg s'(f)\);
   (b) \(\forall f \in e_a^+, s'(f)\);
   (c) \(\forall f \in F - (e_a^- \cup e_a^+)\), \(s'(f) = s(f)\).

The request usually associated with a planning problem is to produce a plan, that is a sequence of actions, whose execution allows the system to go from the initial state to a state that satisfies the goal conditions. It is easy to show that a planning problem \((F, A, Is, G)\) is equivalent to a CNT \((\mathcal{T}L, C)\) where:

1. a state timeline \(s_T\) of boolean domain is associated with each fluent \(f \in F\), at which we add one event timeline \(act\) of domain \(A \cup \{\bot\}\);
2. \(C\) is made of five sets of constraints:
   (a) an instantaneous state basic constraint \(c^I_f = \langle q^I_f, cd^I_f, sc^I_f, df^I_f \rangle\) is associated with each fluent \(f \in F\), with \(q^I_f = \emptyset\), \(cd^I_f = \emptyset\), \(sc^I_f = \{s,f,0\}\), and \(df^I_f \equiv (s,f,0) \rightarrow f \in Is\), to specify the initial state;
   (b) an instantaneous event constraint \(c^P_{f,a} = \langle qt^P_{f,a}, cd^P_{f,a}, sc^P_{f,a}, df^P_{f,a} \rangle\) is associated with each action \(a \in A\) and each fluent \(f \in p_a\), with \(qt^P_{f,a} = [\emptyset]\), \(cd^P_{f,a} = [true]\), and \(\forall i \in I, sc^P_{f,a}(i) = (act_i, s,f,i-1)\) and \(df^P_{f,a}(i) = ((act_i = a) \rightarrow s,f,i-1)\), to specify action preconditions;
   (c) an instantaneous transition constraint \(c^E_{f,a} = \langle q^E_{f,a}, cd^E_{f,a}, sc^E_{f,a}, df^E_{f,a} \rangle\) (resp. \(c^E_{f,a} = \langle qt^E_{f,a}, cd^E_{f,a}, sc^E_{f,a}, df^E_{f,a} \rangle\)) is associated with each action \(a \in A\) and each fluent \(f \in e_a\) (resp. \(f \in e_a^-\)), with \(qt^E_{f,a} = qt^E_{f,a} = [\emptyset]\), \(cd^E_{f,a} = cd^E_{f,a} = [true]\), and \(\forall i \in I, sc^E_{f,a}(i) = sc^E_{f,a}(i) = [act_i, s,f,i]\), and \(df^E_{f,a}(i) \equiv ((act_i = a) \rightarrow s,f,i)\) (resp. \(df^E_{f,a}(i) \equiv ((act_i = a) \rightarrow s,f,i)\)), to specify negative (resp. positive) effects;
   (d) an instantaneous transition constraint \(c^N_{f,a} = \langle qt^N_{f,a}, cd^N_{f,a}, sc^N_{f,a}, df^N_{f,a} \rangle\) is associated with each action \(a \in A\) and each fluent \(f \in F - (e_a^- \cup e_a^+)\), with \(qt^N_{f,a} = [\emptyset]\), \(cd^N_{f,a} = [true]\), and \(\forall i \in I, sc^N_{f,a}(i) = [act_i, s,f,i-1, s,f,i]\), and \(df^N_{f,a}(i) \equiv ((act_i = a) \rightarrow s,f,i-1)\), to specify null effects;
   (e) a unique instantaneous state constraint \(c^G = \langle qt^G, cd^G, sc^G, df^G \rangle\), with \(qt^G = [\emptyset]\), \(cd^G = [true]\), and \(\forall i \in I, sc^G = (F(G))\) (if \(F(G)\) is the set of fluents involved in \(G\)), and \(df^G(i) = \wedge_{g \in G} \neg c\), to specify goal conditions.

Job-shop scheduling

In the scheduling domain, there is no reference framework similar to the STRIPS framework used in planning. There are only problems of very different kind. We focus here on a limited version of the so-called job-shop scheduling, one of the most classical scheduling problems, defined by a finite set \(T\) of tasks with, associated with each task \(t \in T\), a duration \(du_t\), an earliest start time \(es_t\), and a latest end time \(le_t\). One assumes that all the tasks must be performed and that they all require a common non sharable resource: two tasks cannot use this resource at the same time. Let \(nt = |T|\) be the number of tasks. This problem can be modelled by a finite CNT \((\mathcal{T}L, C)\) of length \(l = 2 \cdot nt\) where:

1. one state timeline \(ct\) of domain \(T \cup \{\emptyset\}\) represents the currently active task, with 0 representing the absence of currently active task (there is no need for any event timeline);
2. a constraint \(c_t = \langle qt_t, cd_t, sc_t, df_t \rangle\) is associated with each task \(t \in T\), with \(qt_t = [\emptyset]\), \(cd_t = [true]\), \(\forall i \in I, sc_t(i) = (ct_t, t, t+1)\), and \(df_t(i) = ((ct_t = t) \land (es_t \leq t_1 \leq t+1 \leq le_t) \land ((t+1-t_1) = du_t)\).

What remains to be done

The main result of this article is the proposal of a framework which, the first time as far as we know, allows discrete event dynamic systems (from automata and Petri nets to planning and scheduling) to be modelled in a uniform way using the basic notion of constraint.

The first task is to assess further the modelling power of the proposed framework, by addressing other frameworks such as temporal logics [Pnueli 1977], situation calculus [Levesque et al. 1997], or timed automata [Alur & Dill 1994], as well as various real-world problems.

When timelines are finite (and thus the set of involved variables), this framework allows situations tracking, validation or decision problems to be cast uniformly as CSP or QCSP and solved using any CSP or QCSP solver: CSP for
situation tracking, validation, and optimistic decision problems, and QCSP for pessimistic decision ones. In such a setting, it would be useful to develop or to adapt constraint propagation mechanisms associated with the most useful constraints we identified, which can be seen as global constraints. Beyond, it will be necessary to explore ways of answering requests on infinite timelines, as this is done with automata, Petri nets, and also planning problems.

About possible extensions, a first one would consist in going from hard constraints (which are used here to model possible/impossible facts as well as hard requirements) to soft ones, in order to represent and reason on plausibility and utility degrees, as done in [Pralet, Verfaillie, & Schiex 2006]. This would allow us to capture for example Markov Decision Problems (Puterman 1994) and probabilistic planning [Kushmerick, Hanks, & Weld 1995]. A second orthogonal extension would consist in relaxing the assumption that a state variable remains constant between two successive instants in a timeline and in considering linear, monotonic, or other evolutions (Trinquart & Ghallab 2001). Finally, a third one could relax the assumption of a total order between instants in timelines.

Acknowledgements

This work has been done thanks to the CNES-ONERA-LAAS AGATA project (Autonomy Generic Architecture : Tests and Applications; see http://www.agata.fr), whose aim is to develop technical tools allowing space system autonomy to be improved. It took advantage of the feedback from many people involved in the project, we would like to thank.

References


When the domains of some temporal or atemporal variables are continuous, the resulting CSP/QCSP problems are hybrid discrete/continuous.
Author Index

Abril, Montserrat 6
Arangu, Marlene 68
Barber, Federico 6
Beck, J.Christopher 38
Besse, Camille 14
Cesta, Amedeo 45
Chaib-draa, Brahim 14
Falda, Marco 22
Garrodo, Antonio 30, 68
Kovács, András 38
Lemaître, Michel 84
Oddi, Angelo 45
Onaindia, Eva 30, 68
Perron, Laurent 52
Policella, Nicola 45
Pralet, Cédric 84
Renze Steenhuisen, J. 76
Rossi, Francesca 22
Salido, Miguel A. 6, 60
Sapena, Oscar 30, 68
Shaw, Paul 52
Smith, Stephen F. 45
Venable, K. Brent 22
Verfaillie, Gérard 84
Vidal, Didier 52
Witteveen, Cees 76