Online-2003
Online Constraint Solving:
Handling Change and Uncertainty

A CP2003 workshop, Kinsale, Co. Cork, Ireland
September 29th, 2003

On-line problem solving is a significant issue for many practical applications, where solutions must be executed as the operating environment changes. Many of the application areas have been tackled by constraint based methods, but current constraint solving tools offer little support for on-line problems. Possible enhancements could include rapid reaction to problem changes, prediction of future changes and contingent solutions, time guarantees or exploitation of known time limits.

This workshop aims to bring together researchers interested in these topics, to consider how existing techniques can be enhanced, and to explore combinations of different techniques. The workshop will be of interest to those modelling and solving real world problems, to those interested in theoretical issues in constraint solving, and also to members of the Uncertainty in AI and Planning and Scheduling communities. The most recent antecedents of this workshop were the CP2001 workshops OLCP'01 (Online combinatorial problem solving and Constraint Programming) and CUW'01 (Constraints and Uncertainty).

All submissions to the workshop were reviewed by at least two referees. Five papers were selected for full presentation, and are contained in these working notes. The workshop will finish with an open discussion session on particular topics in Online Constraint Solving. We welcome you to the workshop, and look forward to a fruitful discussion.

Workshop Organisers

Chris Beck	Cork Constraint Computation Centre, Ireland
Ken Brown	Cork Constraint Computation Centre, Ireland
Gérard Verfaillie	RIA Research Group, LAAS-CNRS, France

Program Committee

Roman Barták	Charles University, Czech Republic
Amedeo Cesta	ISTC-CNR, Italy
Markus Fromherz	Parc, USA
Carmen Gervet	IC-PARC, UK
Simon de Givry	INRA, France
Bill Havens	Simon Fraser University, Canada
Narendra Jussien	Ecole de Mines de Nantes, France
Ralf Keuthen	a.p.solve Limited, UK
Ian Miguel	University of York, UK
David Lesaint	BT Exact, France
Jon Spragg	a.p.solve Limited, UK
Thierry Vidal	ENIT, France
Toby Walsh	Cork Constraint Computation Centre, Ireland

Contents

Constraint Maintenance with Preferences and Underlying Flexible Solution

John Bresina[2] Ari Jonsson[1] Paul Morris[2] Kanna Rajan[2]
1. Research Institute For Advanced Computer Science
2. Computational Sciences Division

NASA Ames Research Center, MS 269-1
Moffett Field, CA 94035, U.S.A.

Abstract. This paper describes an aspect of the constraint reasoning mechanism that is part of a ground planning system slated to be used for the Mars Exploration Rovers mission, where two rovers are scheduled to land on Mars in 2004.
The planning system combines manual planning software from JPL with an automatic planning/scheduling system from NASA Ames Research Center, and is designed to be used in a mixed-initiative mode. Among other things, this means that after a plan has been produced, the human operator can perform extensive modifications under the supervision of the automated system. For each modification to an activity, the automated system must adjust other activities as needed to ensure that constraints continue to be satisfied. Thus, the system must accommodate change in an interactive setting.
Performance is of critical importance for interactive use. This is achieved by maintaining an underlying flexible solution to the temporal constraints, while the system presents a fixed schedule to the user. Adjustments are then a matter of constraint propagation rather than completely re-solving the problem. However, this begs the important question of which fixed schedule (among the ones sanctioned by the underlying flexible solution) should be presented to the user. Our approach uses mutable preferences as a prism through which the user views the flexible solution.

1 Introduction

The Mars Exploration Rovers (MER) mission is one of NASA's most ambitious science missions to date. Two rovers were launched in the summer of 2003 with each rover carrying instruments to conduct remote and in-situ observations to elucidate the planet's past climate, water activity, and habitability.

Science is the primary driver of MER and, as a consequence, making best use of the scientific instruments, within the available resources, is a crucial aspect of the mission. In general, every sol (Martian day, about 24 hours and 40 minutes), telemetry from each rover is received on Earth. Based on the downloaded data, a detailed sequence of commands for the next sol must be constructed, verified, and uplinked to the rover. Thus, a viable sequence that satisfies the mission goals needs to be formulated within tight deadlines. To help address this critical need, the MER project has selected MAPGEN (**M**ixed-initiative **A**ctivity **P**lan **GEN**erator) as an activity planning tool.

MAPGEN combines two existing systems, each with a strong heritage: APGEN [6], the Activity Planning tool from the Jet Propulsion Laboratory and the Europa Planning/Scheduling system [4, 7] from NASA Ames Research Center. The Mixed-Initiative aspect means that after an initial plan has been produced, it undergoes a period of "tweaking" by the human operator. Thus, the system must accommodate change, and must do so rapidly enough for interactive use. As we will see, this is achieved by exploiting an underlying flexible solution in Europa so that fast temporal propagation methods can be used.

Flexible time means that instead of finding a single solution, the Planner preserves maximum temporal flexibility by maintaining a set of solutions that satisfy the constraints. This is represented internally as a Simple Temporal Network [3] (STN). As a result of propagation in the STN, each activity acquires a refined time window for its start time.

One advantage of preserving a flexible set of solutions is that the Planner may adapt to additional constraints by exploiting the flexibility, rather than completely re-solving the problem. However, this has to be reconciled with APGEN, which expects to see a fixed time schedule. Also, many tools associated with APGEN, such as those that do calculations of resource usage, require a fixed schedule of activities. Apart from these pragmatic considerations, direct presentation of temporal flexibility to a plan GUI in a way that is not confusing poses significant problems: it is difficult to provide a visual representation of flexibility and temporal relations between activities in a way that does not obscure the display.

The approach we take is to present a single solution to the user in the APGEN GUI, while the Planner maintains the flexible set of solutions as a backup. This raises the issue of determining which fixed schedule to present to the user. In the remainder of the paper, we address this as follows. In section 2, we discuss a natural condidate for a fixed-time schedule, but show that this may violate user preferences. Next, in section 3, we consider constraints and preferences in more detail, and argue that the complexity and transitory nature of the preferences makes it impractical to model them explicitly. The alternative chosen here is to allow the human operator to modify the plan in a way that incorporates his or her implicit preferences. Then, in section 4 we consider an update algorithm that respects this approach. To avoid unnecessarily clobbering the user modifications, the automated system adopts a policy of minimal change (which may be viewed as a surrogate set of preferences). In section 5, we conclude.

2 Earliest Time Solution

The theory of Simple Temporal Networks guarantees that a solution is obtained by assigning to each event the earliest time in its time window. This seems like an obvious candidate for the solution to be presented to APGEN. However, this creates certain undesirable artifacts.

Consider for example an activity that has a flexible start time and flexible duration, but the end time is fixed by a constraint. The earliest time solution will cause the duration to be stretched to its maximum extent, which may not be what the user wants.

In our application, the durations are generally not flexible. Nevertheless, more subtle forms of this problem can occur, as indicated by the following example. The most critical time for the solar-powered Mars Rover in terms of energy is often in the early morning period before the batteries have fully recharged. The CPU is a primary user of energy, and it is required to be on to enable most of the activities. Thus, one way of economizing on energy use is to have greater overlap between CPU-using activities. However, if two overlapping activities are such that the later one is "anchored" in time, while the earlier one is flexible, then they will tend to be pulled apart by the earliest time solution, thus increasing the energy demand.

In the above example, the requirement that fixes the time of an activity is an explicit temporal constraint, and the overlap restriction is a derived preference that may arise from resource limitations. While the earliest time solution may be acceptable as a general default, interactions between constraints and preferences may require departures from this to satisfy specific user needs.

3 Constraints and Preferences

In this application, there is a variety of constraints and preferences that arise from engineering restrictions and scientific need, many of which may not be recognised until specific circumstances arise in operation.

The explicit temporal constraints fall into three categories: *model* constraints, *daily* constraints, and *expedient* constraints. The model constraints encompass definitional constraints and some flight rules. For example, the decomposition of activities into sub-activities specifies temporal relations between the parent and its children. Some activities might be restricted to the day or the night. The daily constraints comprise "on the fly" temporal relations between elements of scientific observations, depending on what scientific hypotheses are being investigated. For example, an image may be taken before using a specific instrument in some circumstances, but not in others. The expedient constraints are those imposed by the Europa planner to guarantee compliance with some higher level constraint that cannot be directly expressed in an STN. For example, a flight rule might specify that two activities are mutually exclusive (such as taking a picture while the rover is moving). This is really a disjunctive constraint, but the planner will satisfy it by placing the activities in some arbitrary order. This has important implications for the tweaking process: the operator may wish to reverse the arbitrary order selected by the planner.

In general, the temporal constraints cover the gamut of those expressible in an STN, including absolute upper and lower time bounds, precedences with quantitative modifiers, relative and absolute deadlines, etc.

There are also preferences that arise from varied sources. Some are based on engineering or scientific considerations such as desiring calibrations to be close to measurements, or wanting separate observations to occur in similar lighting conditions. Perhaps most are derived from the need to solve problems related to resources. In general, the tweaking process is driven by a desire to fit as much "science" as possible into the plan, while steering it on a course that avoids running aground on competing resource limitations. The planner has a limited ability to automatically tweak a plan to try to resolve a

battery energy shortfall, for example by increasing activity overlap (thus reducing cpu time), but most tweaks are performed by the human operator.

Many of the resource calculations are complex. For example, they may involve thermal modelling performed by legacy software. There are often complex options available for reducing usage. For example, in imaging, there are several ways of reducing data volume, such as reducing resolution, or using fewer filters. Choosing between these may require human-level scientific judgement. This means that many of the preferences have an ephemeral nature driven by short-term solutions to transient problems.

These considerations rule out formal modelling of most preferences and dictate the need for a process of informal tweaking by a human operator. The preferences are implicit in the modifications made during this period. However, the modifications interact with the hard constraints discussed above. The automated system must prevent these from being violated. Within this framework, a policy of minimal change provides a reasonable approach for respecting the implicit preferences.

A dramatic illustration of the need for the minimal change occurs when switching from a native APGEN mode, where users are free to modify activities at will, unimpeded by constraints, to the mode where constraints are enforced. To satisfy constraints, some activities must be moved, but arbitrary reorganization of the plan is undesirable.

4 Accommodating Change

Assume that a plan has been produced, and no preferences have yet been expressed to modify the solution. Then the initial solution presented is the earliest time one discussed earlier. During the subsequent tweaking phase, MAPGEN provides a GUI feature, called *constrained move*, that allows dragging an activity to a new location. When the mouse button is released, other activities are also moved to maintain the integrity of the constraints. For example, the moving activity may "push" other activities ahead of it because of precedences established by the user or the planner.

This raises an issue with respect to the expedient constraints. Since these arise from disjunctive constraints that could be satisfied by different arbitrary choices, a mode is provided in which the expedient constraints are relaxed. This allows moved activities to pass over intervening activities that would otherwise be pushed ahead because of expedient constraints. When this relaxed mode is exited, there is a need to re-establish constraints in a way that minimizes the disturbance to the existing plan. A similar need arises when passing from the native APGEN mode to the constraint-maintenance mode. Also, the input files presented to MAPGEN are implicitly in the APGEN mode, and require a similar assimilation to the constraint-maintenance mode.

In this section, we describe the algorithm that is used to modify the solution presented to APGEN by the Europa system. In this interactive application, efficiency considerations seem to rule out the seeking of true optimality (even the tractable kind discussed in [5]). Instead, we have adopted a greedy algorithm that locally minimizes the amount of change from the existing positions of activities.

It is convenient to use a special set of unary singleton constraints to store the current positions of the start and end times of activities. Then the algorithm for updating after a constrained move can be outlined as shown in figure 1.

```
1. Save all the current positions in a temporary list.
2. Remove all the current position constraints and repropagate.
3. For each saved position t of timepoint x  do
      if t is within the STN bounds for x then
         add a position constraint setting x to t
      else if t < the lower bound for x then
         add a position constraint setting x to the lower bound
      else if t > the upper bound for x then
         add a position constraint setting x to the upper bound
      Propagate the effect of the new constraint;
```

Fig. 1. Constrained Update Algorithm

We see that each step that reinstalls a position constraint tries to minimize the departure from the previous position while maintaining consistency. However, the greedy nature of the algorithm means that the order in which activities are considered may affect the outcome. For example, suppose that activity A is constrained to end before activity B starts. If an APGEN file is loaded where activity A is initially simultaneous with B, then one of A or B must be moved. Which of these occurs will depend upon the order in which A and B are considered for the position update in step 3.

The algorithm for updates when exiting a relaxed mode is similar, except that the relaxed constraints are reimposed after step 2. In the case of expedient constraints, the arbitrary planner choices for resolving the disjunctions are subject to change to reflect the saved positions of the timepoints as much as possible.

There are certain situations in which the user needs to ensure that a particular activity prevails in the update lottery. For example, after a constrained move, clearly the activity that is moved should be held to its new position. This is easily done by considering it first. (The new position is guaranteed to be within the STN bounds because a visual indication of these bounds is given during the move, and attempts to move the activity outside that range are ineffective.)

For more general situations, a *pinning* mechanism is provided that allows the user to lock specified activities at their current positions. This is achieved by applying additional constraints. There is a visual indication of which activities are pinned, and they can be unpinned on request. (Certain engineering activities, such as generally immutable communication windows, are pinned by default.)

Note that step 2 of the algorithm requires a repropagation of the network after deletion of constraints. In general, propagations of an STN after deletions are more costly than the simple incremental propagations that occur after additions. However, in the constrained-move case, the deletions are from a consistent state, which means a solution to the STN is already known. This allows us to exploit a trick from Johnson's algorithm [2] so that the $O(N \log N)$ Dijkstra algorithm can be used to update the flexible set of solutions, rather than the more costly $O(EN)$ Bellman-Ford (where N is the number of nodes and E the number of edges in the STN). The trick involves using the known solution to form a new network with non-negative weights that has the same shortest paths as the original network. This can then be used to guide the propagation

in the original network. (As it turns out, even a near-solution provides excellent guidance for the propagation, so this technique is also helpful in repropagation following deletions from an inconsistent state.)

With typical networks not exceeding 1000 nodes, the propagation, using the above mechanism, appears instantaneous to the interactive operator. (Note that if even greater speed were needed, elaborate techniques for incrementality after deletions [1] are available in the literature, but they were not needed in our case.)

When switching from relaxed mode to strict mode (see above), it is possible for the current set of constraints (including pins) to be inconsistent. If inconsistency is encountered during the greedy update, the system removes the most recent implicated activity from the plan, and places it in a temporary storage area called the *hopper*, where it can be inspected by the operator, and possibly reinserted into the plan after further modifications.

5 Closing Remarks

We have discussed an application to assist in ground planning for the MER mission to Mars, and have focused on accomodating change in the context of an interactive tweaking session following automatic plan generation. The modifications provide a way of incorporating implicit preferences in the solution while respecting hard constraints.

Since the mission is still ongoing, this is very much a work in progress. Usability and performance considerations are paramount, and continuing operational readiness tests provide feedback that help to refine the system. Hopefully, a period of reflection following the mission may lead to insights that address these issues at a deeper level.

Acknowledgement: We thank the referees for their useful suggestions.

References

1. R. Cervoni, A. Cesta, and A. Oddi. Managing dynamic temporal constraint networks, in artificial intelligence planning systems. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, 1994.
2. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT press, Cambridge, MA, 1990.
3. R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, May 1991.
4. Ari K. Jonsson, Paul H. Morris, Nicola Muscettola, Kanna Rajan, and Benjamin D. Smith. Planning in interplanetary space: Theory and practice. In *Artificial Intelligence Planning Systems*, 2000.
5. L. Khatib, P. Morris, R. Morris, and B. Venable. Tractable pareto optimization of temporal preferences. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, Acupulco, Mexico, 2003. Morgan Kaufmann, San Francisco, CA.
6. Maldague, Ko, Page, and Starbird. Apgen: A multi-mission semi-automated planning tool. In *Proceedings of the 1st International Workshop on Planning and Scheduling for Space*, Oxnard, California, 1997.
7. N. Muscettola, P.P. Nayak, B. Pell, and B.C. Williams. Remote agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–48, August 1998.

Execution Monitoring and Schedule Revision for O-OSCAR: a Preliminary Report

Amedeo Cesta and Riccardo Rasconi

Planning & Scheduling Team — http://pst.ip.rm.cnr.it
Institute for Cognitive Science and Technology
National Research Council of Italy
Viale K. Marx 15, I-00137 Rome, Italy
{cesta | rasconi }@ip.rm.cnr.it

Abstract. This paper addresses the problem of maintaining the consistency of a pre-defined schedule during its execution in a real or simulated environment. This issue, referred to as Reactive Scheduling Problem, is known to be inherently difficult due to the usually strict timelines in which the revising procedure is called to react. Schedule revision must be quick, and sometimes solution quality must come as a secondary priority as the execution of the schedule does not allow for time-intensive computations. In this work we present a Schedule Execution Monitor and Control System which seizes upon the O-OSCAR (Object-Oriented SCheduling ARchitecture) scheduling tool, a constraint-based software architecture for the solution of complex scheduling problems. The core solving engine of O-OSCAR is represented by the ISES algorithm (Iterative Sampling Earliest Solutions), a constraint-based method for the solution of the RCPSP/Max problem (Resource Constrained Project Scheduling Problem with Time Windows). We have used the preceeding software architecture as a starting point to develop a Schedule Execution Monitor and Control System, capable of reactively maintaining the consistency of the schedule in spite of possible unexpected events to occur at schedule execution time. This paper describes this new module and the current idea of schedule revision also based on the ISES algorithm.

1 Introduction

The problem we are currently studying concerns how to manage a pre-defined schedule while it is being executed in a real or simulated working environment. When we talk about *"real"* or *"simulated"*, we mean an environment which retains some degree of unpredictability. A schedule (or *solution*) consists of a certain number of activities, possibly aggregated in *jobs*, each of which require some resources in order to be executed.

One key point is the fact that resources are limited in *number* and *capacity*. Hence, the need to find a suitable temporal collocation of all the activities such that no resource conflict arises. A *resource conflict* arises every time one or more activities attempts to use a resource beyond its available capacity. When such a

consistent temporal allocation of all the activities is found, the schedule is said to be *feasible*.

A further level of complexity is introduced by the fact that activities can in general be temporally constrained, either individually or among one another: for instance, some operation in a schedule might be constrained not to start before, or not to finish after, a certain instant; in addition, there might be several *precedence constraints* between any two activities in the schedule: for instance, activity B might not be allowed to start before the end of activity A, and so forth.

The issue of schedule *consistency* has therefore two aspects: on one side, **resource consistency** must be maintained at all times, since it is obviously not possible to perform operations when the necessary resources are not available; on the other side, the schedule temporal constraints, i.e. **release time** constraints, **deadline** constraints, **precedence** constraints as well as others, should be kept satisfied as well, as they constitute an integral part of the schedule specifications. A schedule where all the temporal constraints are satisfied, is said to be **temporally consistent**.

Another key issue is the *quality* of the solutions. During schedule execution, we will be faced with the need of revising the actual solution when its consistency has been spoiled by an exogenous event; the activities of the schedule must be allocated anew in order to re-gain consistency and in most cases it is of great importance to keep the new solution as close as possible to the previous one. In other words, it is desirable that the impact of an unforeseen event on the solution be kept to a minimum. In broad terms, a meaningful quality measure could be determined by the level of *continuity* which we are able to guarantee after the revision of the solutions. Of course, there might be cases where maintaining such a high level of continuity is not so important (as well as cases where it is simply not possible!).

One of the major difficulties when working in real environments consists in counteracting the effects that the possible unexpected events may have on the schedule *in a timely manner*. Schedule revision must be quick, and sometimes solution quality must come as a secondary priority because the execution of the schedule does not allow for time-intensive computations.

There are traditionally two approaches to this problem, the **Predictive** approach and the **Reactive** approach [5]. The first one is based on the synthesis of an initially *robust* schedule, that is, a schedule which is capable to absorb, within a certain limit, the spoiling effects of unexpected disturbances without the need of re-scheduling; the second one, which is the approach we are currently studying, tries to maintain consistency by manipulating the schedule every time it is deemed necessary.

In this work, the occurrence of unexpected events will be simulated by randomly changing the actual "world description" by introducing some *disturbances* taken from a pre-defined set, in an attempt to realize a life-like scenario. A software module has been developed specifically to create and inject such disturbances during the schedule execution, the **Contingency Simulator**. The main

goal of the scheduling system will therefore be the one of representing the possible damages, fire the repair action and continously guarantee the executability of the schedule.

To this aim, an **Execution Monitor** has been developed, which is capable to realize a sort of reactive behaviour and conveniently re-adjust the schedule activities by means of the **ISES** procedure (Iterative Sampling Earliest Solutions) [3], a constraint-based method originally designed to solve the **RCPSP/Max** problem (Resource Constrained Project Scheduling Problem with Time Windows).

The Schedule Execution Monitor has been developed as an integration to the **O-OSCAR** (Object-Oriented SCheduling ARchitecture) tool [2], an existing constraint-based software architecture for the solution of complex scheduling problems. The Contingency Simulator is designed as an external module that brings the constraint-based representation abilities into play.

The paper is organized as follows: Section 2 gives a brief overview of the O-OSCAR architecture and its main components; Section 3 introduces the problem of generating contingencies and representing them within O-OSCAR constraint modeling ability, while the Execution Monitor will be described in Section 4. Some conclusions end the paper.

2 The O-OSCAR Software Architecture

In this section we describe the original O-OSCAR scheduling architecture (Fig.1) upon which the Execution Monitor was developed. The whole system initially implemented under MS Windows, is currently being ported to Linux.

The approach used to tackle the scheduling problem is definitely constraint-based. Constraint satisfaction is exploited both as a representation tool as well as mechanism to guide problem solving. Ancestors of O-OSCAR can be considered for example blackboard based architectures like those described in [11, 9, 7]. Similar is also the approach followed in [1]. Distinctive features in O-OSCAR are the particular emphasis given to the flexibility of the core contraints representation, as well as on the central role played by the ISES algorithm [3].

The kernel of the system is the **Representation Module**. Its task consists of maintaining the description of the world (*Domain Representation*) and the description of the problem to be solved (*Problem Representation*).

- **Domain Representation:** All the relevant features of the world and the rules which regulate its dynamic evolution should be described in a *symbolic language*. It is thanks to this basic knowledge that the system is able to offer services.
- **Problem Representation:** A description of the goals in terms of *desired states* of the world must be given, and the scheduler will try to reach the specified goal states starting from the initial one.

O-OSCAR uses a **Constraint Satisfaction Problem (CSP)** approach [12, 8] as the basic modeling tool for scheduling problems. Therefore, all the information maintained in the Representation Module must be organized in terms of

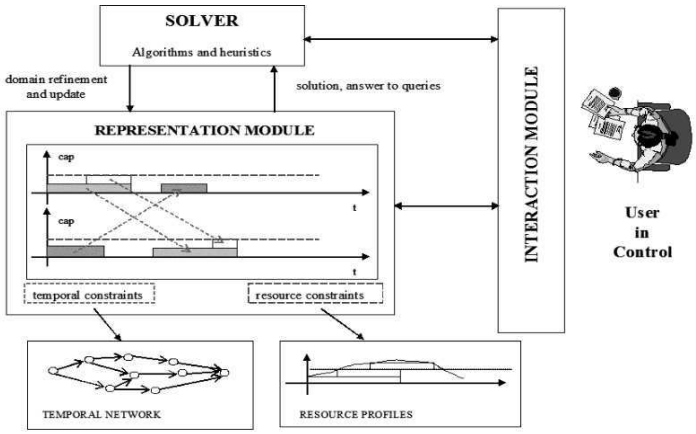


Fig. 1. The O-OSCAR Architecture.

constraints. All the data necessary for the solving process is then stored and kept continously updated in a *Constraint Data Base (CDB)*, which is the core component of any architecture which tackles a CSP. The *CDB* offers an active service which aims at automatically enforcing, *whenever possible*, the satisfaction of the set of constraints which represent both the domain and the problem. To be more specific, it is in charge of the two following aspects:

1. **Domain and Problem Representation:** *The Domain Representation Language* allows the representation of classes of problems as well as the definition of the typical contraints related to each class. In the specific case, O-OSCAR is capable of solving scheduling problems belonging to the **RCPSP (Resource Constrained Project Scheduling Problem)** class, in particular, the RCPSP variant with Generalized Precedence Relations (RCPSP/Max) [10]. *The Problem Representation Language* consists of a set on constraints specifying the activities and their constraint requirements as specified by the RCPSP/max characteristics.
2. **Solution Representation and Management:** The CSP approach to problem solving is based upon the representation, modification and maintenance of a solution. The representation of a solution in O-OSCAR is built on top of two specialized **Contraint Reasoners**, each of them is in charge of a particular aspect of the domain. Two fundamental pieces of information are to be maintained: the information on the *temporal features* of the domain and the information about *resource availability*. The constraint reasoners are called to action every time some changes are made to the current solution description.

The two main domain characteristics that need to be supported in the development of the Constraint Data Base for the resource constrained scheduling problems are:

- *quantitative temporal constraints* allowing specification of both minimum and maximum separation constraints;
- *multi-capacity resources*, that is, the ability of dealing with resources with capacity greater than one.

As shown in Fig.1, the Representation Module is endowed with two constraint reasoners which take charge of the two preceeding aspects: the first one, devoted to the temporal constraint management, stores and analyzes the temporal information making use of a *Temporal Network* and solving in every respect a *Simple Temporal Problem* [6]; the second one stores the resource constraints and reinforces resource consistency by dynamically maintaining specific data structures called *Resource Profiles* which keep information about the consumption level of the available resources.

The features described above are hidden to the external user, who is presented a higher level interface, the *Domain Description Language*. This language brings the typical objects involved in a schedule to the user in terms of higher level entities, such as *activities*, *resources*, *constraints* and *decisions*.

A number of active services can be implemented which seize upon the Representation Module, the first being the *Automated Problem Solving* module (the **SOLVER**, in Fig.1).

This module guides the search for a solution and is based upon the **ISES** algorithm (Iterative Sampling Earliest Solutions) [3]. The module is endowed with two main features: (a) an open framework to perform the search for a solution; (b) heuristic knowledge used to guide the search and lower the computational effort. ISES is defined as a *profile based* procedure: it relies on a core greedy algorithm which operates on a temporally consistent solution, detects the resource conflicts using the information stored in the Resource Profiles data structures, and finally attempts to find a new solution that is *resource consistent*, by imposing some additional precedence constraints between the activity pairs which are deemed responsible for the conflicts, thus flattening the resource contention peaks below the maximum capacity level. This algorithm is iterated until either a resource consistent solution is found or a dead-end is encountered. The greedy algorithm is usually run according to some optimization criteria so to eventually obtain multiple, increasingly better solutions. Some degree of randomization is finally injected in the sampling loop to retain the ability to restart the search in the event that an unresolvable conflict is encountered, without incurring the combinatorial overhead of a conventional backtracking search.

A second, very important module which is present in the O-OSCAR architecture is the one which implements a quite complex Graphical User Interface aimed at guaranteeing a friendly User-System interaction. The services offered by this module vary from simple visualization functionalities, to more sophisticated ones, allowing for instance the direct manipulation of the solution by the user. The goal of such a module is to keep the user always aware and *in control*

of the evolving situation so to enhance collaboration and a synergetic interaction between the intuition capabilities and specific knowledge of human beings on one side, and the computational power of the automated system on the other.

The constraint-based representation mechanism offers the invaluable advantage that additional services can be easily added to the system, relying on the same representation. This will become even more clear in the next sections where we will present the Execution Monitor, a module which *closes the loop* with the real world and dispatches the activities for execution. The Execution Monitor module seizes upon the CSP representation as it implements a reactive approach based on schedule repair, by continously updating the CDB (representing the current status of the solution), according to the possible sudden variations of the schedule at execution time.

3 Simulating and Representing Contingencies

The need to simulate a real working environment led us to develop a particular module, that we called the **Contingency Simulator** (**CS**). Its only purpose is to re-create the same conditions of uncertainty which typically affect the real world, in order to test the repair capabilities of the Execution Monitor. As shown in Fig.2, we have added the Contingency Simulator Module as a block external to the system, with the aim of synthesizing the environment where the schedule will be put in execution.

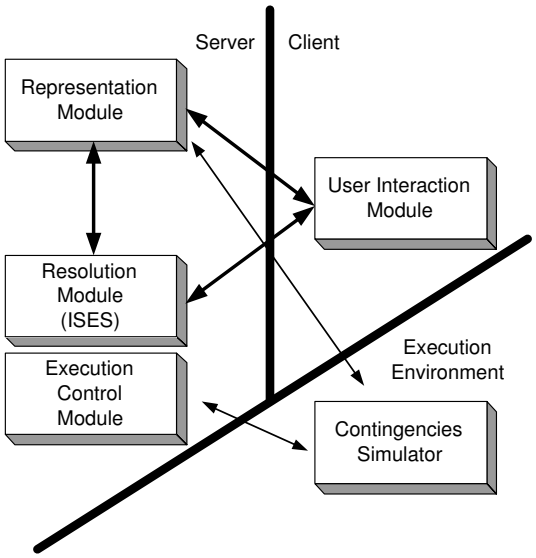


Fig. 2. The O-OSCAR's structure.

The basic job of the CS consists of injecting, during the execution of the schedule, some kind of unexpected events in the environment at random instants. This generates input for the O-OSCAR Representation Module, which stores the world representation.

For instance, at a certain point in execution, the CS may "decide" to suddenly delay an activity, simulating the same kind of incident which could normally occur in everyday's working life. Such delay comes hardly ever without consequences, especially if the schedule has strict time and resource constraints, as it is often the case. Typical consequences triggered by the delay may be for instance the introduction in the schedule of:

- *temporal inconsistency*: the delay may have pushed some activities well beyond some pre-defined temporal deadlines, and the temporal constraint reasoner (see section 2) would promptly assess the situation as not consistent;
- *resource inconsistency*: the delay may have pushed the activity in a time zone where the overall resource requirement exceeds the maximum resource capacity, due to the requests of the other activities which operate in the same interval; as before, the resource constraint reasoner would immediately recognize the conflict.

Three kind of events are currently being simulated by the CS, as they represent a realistic set of probable incidents to occur, for instance, in a real workflow.

Delays on the activities. As mentioned above, the Contingency Simulator may induce a sudden and unexpected time shift on one activity. This is represented within O-OSCAR by inserting a new *precedence constraint* between a particular time point (namely, the *origin* of the temporal network) and the *start time* of the delayed activity. It is worth mentioning here, but it will be reprised later, that the system is endowed with ad-hoc primitives which allow the dynamic insertion and deletion of a number of temporal constraint in and from the schedule. The new insertion is reflected in the insertion of a new constraint between two *time points* at Temporal Network level. Such change is immediately propagated in the network by the proper constraint reasoner. If temporal consistency is not spoiled, we will obtain a new collocation of the activities in the schedule; in the opposite case, the schedule will be found *overconstrained*, meaning that the occurred delay has gone beyond some pre-existing temporal boundaries. As previously stated, resource consistency will then have to be checked as well, and this is when the Execution Monitor revision action may come into play.

Variations on activity durations. Next exogenous event that we have modeled is the change of duration of an activity. This change is represented by substituting the activity with a new one having the same characteristics as for resource requirements, but of course different duration (which may not necessarily be *larger*). As with temporal constraints, proper primitives have been designed which allow the dynamic insertion and deletion of schedule activities.

Again, such a change in the schedule triggers a new propagation in the underlying temporal network to assure that no pre-defined time constraints between any two time points are violated by the substitution.

If the difference ΔT in the activity duration is positive, then resource consistency will again have to be checked, exactly as in the previous case; in case ΔT should be negative (the activity will last shorter than expected), no resource conflict may ever arise, yet a revision of the schedule is still recommended so to take profit of possible *opportunities*, created by the sudden vacancy of resource usage.

Resource breakdowns. The last event we are going to model is the unexpected loss of some resource. In case of such an occurrence, it is very likely that a resource conflict may arise, therefore the need of a quick re-schedulation of the activities. Our system can handle *multicapacity* resources, that is, resources having capacity greater than one: therefore the CS should be able to simulate also the *partial* loss of a resource capacity, (temporary or definitive). For example, assuming the presence of three identical trucks as resources in a schedule (resource type: *truck*, resource capacity: *three*), one might want to re-create the loss of just one truck.

O-OSCAR models resource breakdown by the simple insertion of a new *ghost* activity with the only aim to add another source of contention in the schedule. The ghost activity will make use of the resources which have to be collapsed, *of exactly the capacity that has to be collapsed*. In other words, the condemned resource will be "eaten out" by the ghost activity, obtaining as an overall effect, its partial or total breakdown.

An important issue has to be raised at this point: the *ghost* activities which are added by the CS to simulate resource unavailability are activities which do not belong to the schedule and as such, must not participate to any possible schedule revision process. As we will see in the next section, the ISES procedure is not capable of distinguishing ghost activities from the ordinary ones, so the only way to exclude the former from being re-scheduled is to "anchor" them, once and for all, to the time interval where they have been placed initially. This is achieved by inserting a so called **Fixtime constraint** (see section 4) and relating it to the start time of each ghost activity. The Fixtime constraint basicly imposes a rigid distance between any two time points; any attempt to change their mutual distance will be forbidden by the time constraint reasoner as it may cause a violation of the Temporal Network consistency. In our specific case, the Fixtime constraint imposed on a ghost activity simply fixes the distance between the time point *origin* of the Temporal Network and the ghost activity start time.

As it will be shown in detail in the next section, the expressiveness of the constraint-based design of O-OSCAR has been fully exploited in the development of the CS module. A set of higher level primitives for the insertion/deletion of constraints in/from the schedule has been implemented which guarantees rich and complete representation capabilities. Based on the versatility of O-OSCAR, these primitives not only permit a dynamic management of all kinds of temporal constraints on the basis of a few simple parameters to be supplied by the user, but as will be more clear shortly, they add a further degree of expressiveness to the system by introducing the capability to add or remove new activities on the fly, during schedule execution.

4 The Execution Monitor

Once an initial solution to a given problem is obtained, the **Execution Control** module (see Fig.2) is responsible for dispatching the activities of the plan for execution and detecting the status of both the execution and of the relevant aspects of the world.

As explained in the previous section, the detected information is used to update the CDB in order to maintain the world representation perfectly consistent with the evolution of the real environment; the main issue is that updating the data stored in the Representation Module in accordance to the information gathered from the environment may introduce some inconsistency in the schedule representation.

We have implemented an Execution Monitor which reacts to these inconsistencies as they are detected, namely attempting to take the schedule back to a consistent state, so as to keeping it executable.

The repair action is performed by exploiting the capabilities of the ISES algorithm, which is used as a "black box"; in other words, schedule revisions are approached as a *global* re-scheduling actions, without focusing on a particular area of the solution, as realized with different approaches, e.g., [11].

Updating the schedule representation. Our global approach requires some preventive action to be taken before the ISES procedure is fired, in order to have the necessary control on schedule repair choises. In other words, we can guide the revision process by preventively constraining the activities, depending on the strategies we want to realize. A number of primitives have been developed which make the dynamic insertion and deletion of temporal constraints possible.

At present time these primitives handle the following constraints:

1. **Precedence Constraint**: imposes a temporal relationship between two activities; in the shown example, activity A2 cannot start *before* the end of activity A1 [$\boldsymbol{st2 \geq st1+dur(A1)}$].
   This is achieved by imposing a minimum distance between the start times of the activities A1 and A2, equal to the duration of A1. In this way, we keep the two activities *relatively* separated, so that a temporal shift executed on A1 would immediately induce a shift on A2 as well;
2. **Deadline Constraint**: imposes a time boundary on the activity *end time*; the activity cannot end *after* the deadline **dl** [$\boldsymbol{et \leq dl}$];
   This is achieved by imposing a maximum distance between the time point *origin* and the end time of the activity. In this way, we **do not** force the activity to end at instant $\boldsymbol{dl}$, but we certainly ensure that it will not terminate beyond that point;
3. **Release Time Constraint**: imposes a time boundary on the activity *start time*; the activity cannot start *before* the release time **rt** [$\boldsymbol{st \geq rt}$];
   This is achieved by imposing a minimum distance between the time point *origin* and the start time of the activity. In this way, we **do not** force the activity to start at the instant $\boldsymbol{rt}$, but we certainly ensure that it will not attempt execution before that point;

4. **FixTime Constraint**: similar to the previous one, imposes a more rigid constraint on the activity *start time*; the activity cannot start neither *before*, nor *after* a determined fixed instant **ft** [$\boldsymbol{st = ft}$];
   This is achieved by imposing a minimum and a maximum distance, ***mind*** and ***maxd*** respectively, between the time point *origin* and the start time of the activity, with $\boldsymbol{mind = maxd = st}$. In this way, any attempt to move the activity away from its determined *FixTime* instant will be forbidden by the temporal constraint reasoner.

```
 1 T ⟵ 0
 2 execSched ⟵ schedule0
 3 while(Schedule NOT executed)
 4   ENVIRONMENT SENSING
 5   if (Unforeseen Events)
 6     UPDATE REPRESENTATION
 7     if (NOT Temporal Consistency)
 8       EXIT WITH FAILURE
 9     else if (NOT Resource Consistency)
10           SCHEDULE REVISION
11     if (Conflicts NOT eliminated)
12       EXIT WITH FAILURE
13   else
14     T = T+1
```

Fig. 3. The execution algorithm.

Executing the schedule. The pseudo-code in Fig.3 describes the execution algorithm. At the beginning, the time variable **T** is initialized and so is the variable ***execSchedule*** which refers to the schedule under execution.

At every cycle, the environment is sensed in order to detect any possible deviation between the expected and the actual situation; if unforeseen events have occurred, we update the world representation stored in the CDB, to reflect the new world status.

Next step consist of checking **temporal consistency**, as in the CDB updating process we may have added to the representation some temporal constraints which are in conflict with the existing ones.

If consistency is lost, the algorithm must terminate with failure, as no repair action is possible unless some previously imposed constraints are relaxed; if time consistency is not spoiled, **resource consistency** must be checked as well, because the occurrence of the exogenous event may have introduced some resource conflicts in the schedule, although leaving it temporally consistent.

If no resource conflicts are present, the execution of the schedule may continue; otherwise, a **schedule revision** must be performed, in the attempt to

eliminate resource contention. If the schedule revision process succeeds in eliminating the conflicts, execution may continue; otherwise the algorithm must exit with failure.

4.1 Schedule Revision

Let us take a closer look at the way the activities in the schedule are actually manipulated during execution and repair. As previuosly stated, the approach used in our Execution Monitor can be considered as *global* [4, 13] in that the revision procedure accepts the schedule *as a whole*, tries to solve all the presents conflicts and returns the solution. In other words, the ISES procedure does not make any difference between terminated, started or yet-to-start activities, and has no concept of *time*. As a consequence, the only chance at our disposal to exert some control over the activities is to do it in a preventive way, that is, *before* the ISES procedure begins the manipulation of the schedule.

Such control is necessary for at least two reasons: (a) we want to keep the solutions *physically consistent* at all times; (b) we want to retain the possibility to satisfy a set of *preferences* given by the users.

Let's go deeper in the subject by focusing on some practical issues arising during schedule execution that we have to face in order to obtain meaningful results from the revision pocess. We assume the schedule under execution with *current execution time* $= t_E$.

Physical Consistency. The consistency must be satisfied at all times, because the current solution always embodies the description of a real world situation.

There are many ways in which physical consistency may be spoiled as a result of an inattentive action: for instance, the re-scheduling procedure may try to re-allocate some activities which have already started execution.

Clearly, this represents an inconsistent situation and must be avoided at all costs. The problem is solved by inserting a new **FixTime constraint** for every activity whose start time $\boldsymbol{st} = t_E$. By doing so, we impose a very strict temporal constraint on the activity start time: *all the solutions found by ISES which require a temporal shift of the constrained activity, will be rejected.*

As another example, the re-scheduling procedure may allocate some activities *at the left* of t_E in the temporal axis, which would be equivalent to allocating operations **in the past**. All we have to do in this case is to introduce in the schedule as many **Release Time constraints** as are the activities whose start time $\boldsymbol{st}$ is greater or equal than t_E. In other words, we constrain all the activities which have not yet started, not to begin execution before the current execution time. Again, this does not necessarily mean that these activities will be moved by ISES: anyway, should they be re-allocated, they would certainly be positioned at the right of t_E.

Preferences Management. As anticipated at the beginning of this paper, *Solution Continuity* can be a very important quality measure of the schedule. In many cases it is essential that any revised solution be as close as possible to the last

consistent solution found by ISES; the closer any two solutions are, the higher their level of continuity.

It is in fact desirable (and plausible) that despite the possible exogenous events that may occur during the execution of a schedule, this remains as similar as possible to the initial solution, as it is supposed to be close to the optimal one. Schedule continuity can be controlled by leaving or removing the **precedence constraints** possibly imposed in the last execution of ISES. It is known that ISES resolves the conflicts by inserting a certain number of extra precedence constraints between the activities, in order to separate them in the areas of greater resource contention; these extra constraints are not part of the original problem and are only there to solve a particular resource conflict. In case ISES should be run one more time, one might decide to remove these constraints, counting on the fact that the conflicts re-introduced by this removal will be solved by the next execution of ISES.

Depending on which decision is taken, (removing or leaving these constraints), it has been observed that the new solution shows respectively a lower or higher level of continuity with respect to the old one, obviously due to the different degree of liberty retained by the activities in the two cases. The more constrained the activities are, the lower the possibility that the new solution differs from the old one. Depending on how important continuity is for the problem at hand, one may choose which approach to adopt.

There are instances in which an intensive reallocation of the scheduled activities in response to an exogenous event may represent a serious problem; let's think of a workflow in a manifacturing environment: its execution may involve a great work spent in team organization and manpower management. In such cases, the main interest is to preserve solution continuity should something go wrong, so as to minimize the amount of work which would be necessary to re-organize all the employees' working shifts, not to mention the work to re-organize the raw materials shipment deadlines.

On the other hand, there are working environments (maybe more automated), where not only the re-allocation of scheduled activities does not represent a problem, but where a major re-allocation of the operations, even if caused by an unexpected event, may be considered as an opportunity not to be renounced. In these last cases, solution continuity is not a major concern.

As a last observation, with a proper handling of the temporal constraints it is also possible to bias the schedule in order to satisfy user's preferences; for instance, before schedule revision it could be possible to specify the *degree of mobility* of the activities, such as maximum delays, preferred anticipations, and so on.

By exerting this kind of preventive control, it is possible to express preferences on the behaviour of every individual activity before schedule revision, thus obtaining a solution which best suites the user's desires.

4.2 Current Status

The actual system has been implemented and tested on a preliminary set of Multi-Capacity Job Shop Scheduling benchmark problems of the order of $15 \div 30$ activities. The obtained results are very encouraging: as contingencies of different nature and gravity are injected in the execution environment, the system succeeds in quickly working out a new consistent solution. For example, we tested the system ability to recover from sudden delays of different gravity of one or more activities, at various stages of execution, as well as from sudden resource breakdowns, with partial or total loss of one or more resources, still at various stages of execution.

At the current stage of development we are focusing our attention on the methodologies (constraint posting strategies, general user service design, etc.) more than on true system performance. Resolution speed will be our next objective: the actual implementation of O-OSCAR presents a number of points where computation can be made faster, depending merely on implementing issues. To give an idea of the present performance level, the system succeeds in rescheduling a RCPSP/Max problem with 20 activities and 3 resources in an average time of 190 milliseconds, during schedule execution.

It is worth noting that the nature of the reactive scheduling problem entails the presence of several parameters, and therefore much attention must be paid in order to synthesize a meaningful benchmark. Among the parameters involved are the following:

- temporal separation between the current instant of execution and the sensed conflict;
- type of contingency occurred;
- number of activities affected;
- number of resources affected;
- for each resource, the capacity affected.

Each one of the previous variables may trigger a different response from the system, and in case of multiple occurrences, their mutual timing will be another major source of performance variability. In general a lot of work is still to be done, but the building blocks descibed in this paper were a needed precondition for actually performing such work.

5 Conclusions

In this paper we have presented the current status of the execution monitoring module of the O-OSCAR architecture. The model implements an approach to schedule revision that we call *global reaction approach* to distinguish it from the one followed for example in [11] that coud be called *local reaction approach*. According to this strategy, we perform the rescheduling of the entire set of activities not executed before the current execution time, including those not affected by the exogenous event.

The global approach we have dscribed relies entirely on the solving capabilities of the ISES procedure, coupled with the expressive power of a set of primitives which extensively exploit O-OSCAR representation features. ISES is used as a black box, and this requires a careful preventive action in order to exert some control on the dynamic evolution of the whole system. This control is exerted by means of a skillful use of the above mentioned primitives, which allow to easily manipulate both the activities of the schedule and the time constraints insisting on those activities.

On the opposite, the local reaction methodology seizes on the analysis of the occurred conflicts and on the utilization of specialized metrics in order to execute the most suitable repair action, chosen from a set of pre-defined revision procedures, on the most urgent conflict among those waiting to be attended; whatever the chosen repair method and the chosen conflict, the solution revision will be not be performed on the entire schedule as with the global approach, but on a limited number of activities, namely, those which are deemed to be the most seriously involved in the conflict. The previous step is iterated until all pending conflicts are solved (see [11] for further details).

We believe that our system with its core constraint-based architecture and the primitives at disposal, constitutes a solid framework also for the implementation of a locally reactive scheduling system, as just described. Future developments of our work include the realization of such a alternative approach by means of the existing O-OSCAR building blocks. The idea is to measure the system responsiveness of both approaches and their ability to recover from inconsistent states under different conditions of execution. Among the measures of interest we identify the schedule *makespan* at the end of the execution, the schedule *global lateness* as a weighed average of the individual delays with respect to the initial solution, and of course an appropriate measure of solution continuity, which can be initially taken as a meaningful measure of schedule quality.

Acknowledgements

This research is partially supported by ASI (Italian Space Agency) under project ARISCOM (Contract I/R/215/02) and by MIUR (Italian Ministry of Education, University and Research) under project RoboCare (A Multi-Agent System with Intelligent Fixed and Mobile Robotic Components). The authors are part of the Planning and Scheduling Team [PST] at ISTC-CNR and would like to thank the other members of the team for several technical interactions.

References

1. Beck, J. C., Davenport, A., Davis, E., and Fox, M. S. The ODO Project: toward a unified basis for constraint-directed scheduling. *Journal of Scheduling 1* (1998), 89–125.
2. Cesta, A., Cortellessa, G., Oddi, A., Policella, N., and Susi, A. A Constraint-Based Architecture for Flexible Support to Activity Scheduling. In

Proceedings of 7th Congress of the Italian Association for Artificial Intelligence (2001).
3. Cesta, A., Oddi, A., and Smith, S. F. A Constrained-Based Method for Project Scheduling with Time Windows. *Journal of Heuristics 8*, 1 (2002), 109–135.
4. Church, L. K., and Uzsoy, R. Analysis of Periodic and Event-Driven Rescheduling Policies in Dynamic Shops. *Inter. J. Comp. Integr. Manufact. 5* (1991), 153–163.
5. Davenport, A., and Beck, J. C. A Survey of Techniques for Scheduling with Uncertainty. http://www.eil.utoronto.ca/profiles/chris/chris.papers.html.
6. Dechter, R., Meiri, I., and Pearl, J. Temporal Constraint Networks. *Artificial Intelligence 49* (1991), 61–95.
7. Le Pape, C. Scheduling as Intelligent Control of Decision-Making and Constraint Propagation. In *Intelligent Scheduling*, M. Zweben and S. M. Fox, Eds. Morgan Kaufmann, 1994.
8. Montanari, U. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Sciences 7* (1974), 95–132.
9. Muscettola, N. HSTS: Integrating planning and scheduling. In *Intelligent Scheduling*, M. Zweben and S. M. Fox, Eds. Morgan Kaufmann, 1994.
10. Schwindt, C. Generation of Resource Constrained Project Scheduling Problems with Minimal and Maximal Time Lags. Tech. Rep. WIOR-489, Universität Karlsruhe, 1996.
11. Smith, S. F. OPIS: A Methodology and Architecture for Reactive Scheduling. In *Intelligent Scheduling*, M. Zweben and S. M. Fox, Eds. Morgan Kaufmann, 1994.
12. Tsang, E. *Foundations of Constraints Satisfaction.* Accademic Press, London, 1993.
13. Wu, S. D., Storer, H., and Chang, P. C. One-machine rescheduling heuristics with efficiency and stability as criteria. *Comput. Oper. Res. 20* (1993), 1–14.

Super CSPs

Emmanuel Hebrard, Brahim Hnich, and Toby Walsh*

Cork Constraint Computation Centre
University College Cork
{e.hebrard, brahim, tw}@4c.ucc.ie

Abstract. Fault tolerant solutions [12] and supermodels [8] are solutions with strong properties of stability. In this paper, we study super solutions, the generalization of supermodels to the constraint satisfaction and optimization framework. We explore two different approaches to find super solutions. In the first, we reformulate a constraint problem so that the only solutions are super solutions. In the second, we introduce notions of super consistency and enforce them during search. We also propose a branch and bound algorithm for finding the most robust solution, in case no super solutions exist. Finally, we run extensive experiments to compare the different approaches and study the difficulty of finding super solutions. We show that super MAC, a new search algorithm for finding super solutions outperforms the other techniques.

1 Introduction

Many AI problems may be modelled as constraint satisfaction and optimization problems. However, the real world is subject to change: machines may break, drivers may get sick, stock prices increase or decrease, etc. In such cases, our solutions to the problems may "break". In this context, one may want a solution to be robust, that is able to remain valid despite changes.

Uncertainty and robustness can be incorporated into constraint solving in many different ways. Some have considered robustness as a property of the algorithm, whilst others as a property of the solution (see, for example, dynamic CSPs [1] [7] [10], partial CSPs [5], dynamic and partial CSPs [9], stochastic CSPs [11], and branching CSPs [3]). In dynamic CSPs, for instance, we can reuse previous work in finding solutions, though there is nothing special or necessarily robust about the solutions returned. In branching and stochastic CSPs, on the other hand, we find solutions which are robust to the possible changes. However both these frameworks assume significant information about the likely changes (e.g. the stochastic CSP framework assumes we have independent probabilities for the values taken by the stochastic variables). In this paper we generalize a definition of solution robustness introduced in SAT [8] to constraint programming. This definition allows us to estimate the robustness of solutions without any additional knowledge.

* All authors are supported by the Science Foundation Ireland.

Solution stability[1] is the ability of a solution to share as many values as possible with a new solution if a change occurs. For example, a stable solution in a trip planning problem would not require cancelling a flight because of a train drivers' strike: the plan should change locally and in small proportion. Where large changes to a solution introduce additional expenses or reorganization, stability is valuable. Moreover, stability can help us find a new solution. Stability can then be seen as a particular form of robustness.

Fault tolerant solutions [12] and supermodels [8] are examples of solutions that exhibit strong properties of stability: a solution is fault tolerant if any of its values can be replaced by another one. For each part of our trip we need at least two different ways to get from one point to another (we can replace the train by a bus). Supermodels are models of SAT formula that can be repaired once a small number of variables have changed by changing only a few other variables. Supermodels are a powerful way to capture robustness and stability of solutions. Supermodels are computed offline, in advance of any changes. A supermodel guarantees the existence of a reasonably small repair in case of a small change in the future. Supermodels do not require any particular knowledge about future changes. However, supermodels have only been studied for SAT problems. In this paper, we generalize the concept of supermodels to constraint satisfaction problems (CSPs). We conjecture that constraint programming is in many ways a better framework for supermodels: they will be more likely, and they will more likely be useful. The definition of supermodels given in [8] has to be modified to deal with CSPs. From now on, we will refer to supermodels for SAT problems, whereas *super solutions* will denote stable solutions to CSPs. Note that our definition of super solutions (section 2) reduces to the definition of supermodels if the SAT variables are considered as CSP variables with binary domains.

2 Super Solutions

Supermodels were introduced in [8] as a framework to measure inherent degrees of solution stability. An (a, b)-supermodel of a SAT problem is a model (a satisfying assignment) with the additional property that if we modify the values taken by the variables in a set of size at most a (breakage variables), another model can be obtained by flipping the values of the variables in a disjoint set of size at most b (repair variables). A necessary but not sufficient condition that need to be satisfied in order to find a supermodel is the absence of backbone variables. A *backbone variable* is a variable that takes the same value in all solutions. The presence of a backbone variable in a SAT problem makes it impossible to find any (a, b)-supermodels as that particular variable has no alternative.

There are a number of ways we could generalize the definition of supermodels from SAT to constraint satisfaction as variables now can have more than two values. A break could be either "losing" the current assignment for a variable and then freely choosing an alternative value, or replacing the current assignment

[1] sometimes also referred to as "similarity" in the literature

with some other value. Since the latter is stronger and therefore less useful, we propose the following definition.

Definition 1. *A solution S to a CSP is (a, b)-super solution iff the loss of the values of at most a variables in S can be repaired by assigning other values to these variables, and modifying the assignment of at most b other variables.*

A number of properties follow immediately, for example, a (c, d)-super solution is a (a, b)-super solution if ($a \leq c$ or $d \leq b$) and $c + d \leq a + b$,

We will focus mostly on $(1, 0)$-super solutions in the rest of the paper. They are called *fault tolerant solutions* and described in [12]. Deciding if a SAT problem has an (a, b)-supermodel is NP-complete [8]. It is not difficult to show that deciding if a CSP has an (a, b)-super solution is also NP-complete, even when restricted to binary constraints.

Theorem 1. *Deciding if a CSP has an (a, b)-super solution is NP-complete for any fixed a.*

Proof. To see it is in NP, we need a polynomial witness that can be checked in polynomial time. This is simply an assignment which satisfies the constraints, and, for each of the $O(n^a)$ (which is polynomial for fixed a) possible breaks, the $a + b$ repair values.

To show completeness, we show how to map a binary CSP onto a new binary problem in which the original has a solution iff the new problem has an (a, b)-supersolution. We duplicate the domains of each of the variables, and extend the constraints so that the behave equivalently on the new values. For example, suppose we have a constraint $C(X, Y)$ which is only satisfied by $C(m, n)$. Then we extend the constraint so that is satisfied by just $C(m, n)$, $C(m', n)$, $C(m, n')$ and $C(m', n')$ where m' and n' are the duplicated values for m and n. Clearly, this binary CSP has a solution iff the original problem also has. In addition, any break of a variables can be repaired by replacing the a corresponding values with their primed values (or unpriming them if they are already primed) as well as any b other values.

3 Motivational Example

The approach taken in this paper, whilst it concerns repairs, is a proactive approach. A super solution is a solution to the deterministic, regular, CSP which we expect may change before we come to apply the solution. The changes occur *after* we have found a solution and must then be tackled. We aim to ensure that any break (loss of one value) will be repairable if it eventually occurs.

Let us consider the following CSP: $X, Y, Z \in \{1, 2, 3\} \quad X \leq Y \wedge Y \leq Z$
The solutions to this CSP are shown in Figure 1, as well as the subsets of solutions that are $(1, 1)$-super solutions and $(1, 0)$-super solutions for this problem.

The solution $\langle 1, 1, 1\rangle$ is not a $(1, 0)$-super solution. The reason is that if X loses its value 1, we cannot find a repair value for X that is consistent with Y

solutions	$(1,1)$-super solutions	$(1,0)$-super solutions
$\langle 1,1,1\rangle$, $\langle 1,1,2\rangle$	$\langle 1,1,2\rangle$, $\langle 1,1,3\rangle$	$\langle 1,2,3\rangle$
$\langle 1,1,3\rangle$, $\langle 1,2,2\rangle$	$\langle 1,2,2\rangle$, $\langle 1,2,3\rangle$	$\langle 1,2,2\rangle$
$\langle 1,2,3\rangle$, $\langle 1,3,3\rangle$	$\langle 1,3,3\rangle$, $\langle 2,2,2\rangle$	$\langle 2,2,3\rangle$
$\langle 2,2,2\rangle$, $\langle 2,2,3\rangle$	$\langle 2,2,3\rangle$, $\langle 2,3,3\rangle$	
$\langle 2,3,3\rangle$, $\langle 3,3,3\rangle$		

Fig. 1. solutions, $(1,1)$-super solutions, and $(1,0)$-super solutions for the problem $X \leq Y \leq Z$.

and Z because neither $\langle 2,1,1\rangle$ nor $\langle 3,1,1\rangle$ are solutions to the problem. Also, solution $\langle 1,1,1\rangle$ is not a $(1,1)$-super solution because when X loses its value 1, we cannot repair it by changing the value assignment of at most another variable, i.e., there exists no repair solution when X breaks because none of $\langle 2,1,1\rangle$, $\langle 3,1,1\rangle$, $\langle 2,2,1\rangle$, $\langle 2,3,1\rangle$, $\langle 2,1,2\rangle$, and $\langle 2,1,3\rangle$ is a solution to our problem. On the other hand, $\langle 1,2,3\rangle$ is a $(1,0)$-super solution because when X breaks we have the repair solution $\langle 2,2,3\rangle$, when Y breaks we have the repair solution $\langle 1,1,3\rangle$, and when Z breaks we have the repair solution $\langle 1,2,2\rangle$. We therefore have a theoretical basis to prefer the solution $\langle 1,2,3\rangle$ to $\langle 1,1,1\rangle$, the former being more "robust" or "stable". Note that all algorithms introduced in this paper provide offline (that is, in advance) the repairs as well as the solution. Hence finding and applying the repairs online takes constant time.

The way a given problem is modelled influences the super solutions. For instance, consider the encoding in SAT of this problem. One way to encode this is to add a Booloean variable x_i for every value i that X can take, $x_i = True$ means that $X = i$. In our case, $\{x_1, x_2, x_3\}$, $\{y_1, y_2, y_3\}$ and $\{z_1, z_2, z_3\}$. However, such an encoding has no $(1,0)$-supermodel. Any variable y_i standing for an assignment of y is in conflict with at least one other assignment on x or z. Moreover, one y_i must be set to $True$, since any solution gives a value to y. Therefore the variable in conflict with y_i must be set to $False$. If the assignment of this variable is modified, i.e, flipped to $True$, then at least y_i must be reassigned to $False$. Intuitively, in any encoding, the likelihood of finding an (a,b)-super solution will decrease with the number of variables and increase with the domain size. Moreover, the meaning of a super solution depends also on the model. For example, if a variable is a country and a value is a colour, the loss of a given value is equivalent to the loss of the given colour. On the other hand if every possible colouring of that country is encoded by a Boolean variable, the loss of a value means either that the colour is now forbidden or that this colour *must* be used. The CSP framework gives more freedom to choose what variables and values stand for, and therefore what being a super solution means.

4 Reformulation Approach to Finding Super Solutions

One possible approach to finding super solutions is to add further variables and constraints to the CSP that would eliminate those solutions that are not

super solutions. In [12] a definition of *fault tolerant solutions* is given which matches exactly the definition of $(1,0)$-super solutions: two reformulations of CSP are given in [12] such that any solution of the reformulation corresponds to a fault tolerant solution of the original CSP. In the following subsections, we review those reformulation approaches and propose a new one, which we call the *cross-domains representation.*

4.1 Boolean Reformulation

The first approach in [12], associates a Boolean variable x_v to each value v of each variable X in a given CSP. Assigning this variable to 1 corresponds to the assignment $X = v$ in the CSP. Every disallowed tuple $\neg(X = v, Y = w)$ translates into the conflict clause which forbids the assignment (1,1) for the two corresponding variables. Finally, whereas in the original CSP any variable must implicitly be given exactly one value, here exactly *two* variables must be satisfied for every CSP variable. This model allows only fault tolerant solution, but not *all* of them and it is shown through the following CSP:

$$X = [1,2], Y = [1,2], X + Y < 4$$

This CSP translates into the following Boolean CSP (or SAT problem):

$$x_1, x_2, y_1, y_2 \in \{0,1\} \ (1)$$
$$(x_1, x_2) \in \{(1,1)\}, (y_1, y_2) \in \{(1,1)\} \ (2)$$
$$(x_2, y_2) \in \{(0,0), (0,1), (1,0)\} \ (3)$$

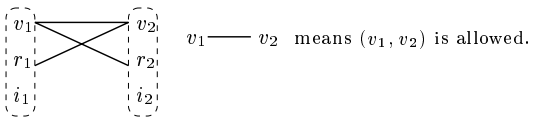
In (1) the Boolean variables associated to the different assignments are given. In (2) exactly two of the Boolean variables must be true for each corresponding CSP variable, while in (3), the only disallowed tuple $\langle X = 2, Y = 2\rangle$ is encoded. The solution $\langle 1,1\rangle$ is a fault tolerant solution of the original CSP, since for X or Y, the value 2 can replace the value 1. However, this Boolean CSP has no solution. This reformulation does not therefore give all the fault tolerant solutions.

4.2 Adding Extra Variables and Constraints

The second approach proposed in [12] simply duplicates the variables. The additional variables have the same domain as the original variables, and are linked with the same constraints to the same neighbourhood. A not_equal constraint is also posted between each original variable and its duplicate. The assignment to the original part of the CSP gives then the solution, while the duplicated part gives the repair for each variable. We refer to the reformulation of a CSP P with this encoding as $P+P$.

4.3 Cross-Domains Reformulation

We now present our last reformulation approach. Let $S = \langle v_1, v_2 \rangle$ be a $(1,0)$-super solution on *two* variables X_1 and X_2. If v_1 is lost, then there must be a value in $r_1 \in \mathcal{D}(X_1)$ that can repair v_1, that is $\langle r_1, v_2 \rangle$ is a compatible pair. Symmetrically, there must exists r_2 such that $\langle v_1, r_2 \rangle$ is allowed. Now consider the following subproblem involving two variables:

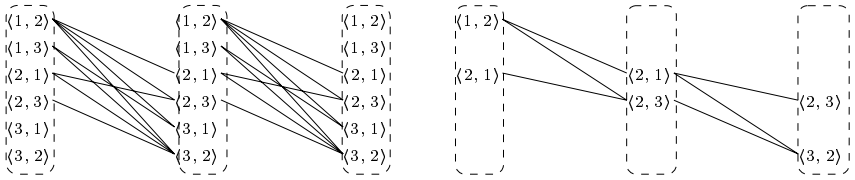


Since it satisfies the criteria above, $S = \langle v_1, v_2 \rangle$ is a super solution whilst any other tuple is not. One may suspect that values r_1, r_2, i_1, i_2 do not participate in any super solution and hence can be pruned. However r_1 and r_2 are *essential* for *providing* support to v_1 and v_2. So, we cannot simply reason about extending partial instantiations of values, unless we keep the information about the values that can be used as repair. So, let us instead think of the domain of the variables as pairs of values $\langle v, r \rangle$, the first element corresponding to the *super value* (which is part of a super solution), the second corresponding to the *repair value* (which can repair the former). We call $P \times P$ the reformulation of a CSP $P = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$ such that any domain becomes its own cross-product (minus the doublons), D_i becomes $D_i \times D_i - \{\langle v, v \rangle | v \in D_i\}$. The constraints are built as follows, two pairs $\langle v_1, r_1 \rangle$ and $\langle v_2, r_2 \rangle$ are compatible iff

- v_1 and v_2 are compatible (the solution must be consistent at the first place),
- v_1 and r_2 are compatible (in case of break involving v_2, r_2 can be a repair),
- v_2 and r_1 are compatible.

The new domain $\mathcal{CD}(\mathcal{X})$ and $\mathcal{CD}(\mathcal{Y})$ of variable X and Y are
$\{\langle v_1, r_1 \rangle, \langle v_1, i_1 \rangle, \langle r_1, v_1 \rangle, \langle r_1, i_1 \rangle, \langle i_1, v_1 \rangle, \langle i_1, r_1 \rangle\}$
$\{\langle v_2, r_2 \rangle, \langle v_2, i_2 \rangle, \langle r_2, v_2 \rangle, \langle r_2, i_2 \rangle, \langle i_2, v_2 \rangle, \langle i_2, r_2 \rangle\}$
and the only one allowed tuple is $S = \langle \langle v_1, r_1 \rangle, \langle v_2, r_2 \rangle \rangle$. The example below shows the cross-domain representation of the CSP given in section 3 ($X \leq Y \leq Z, D(X) = D(Y) = D(Z) = [1,2,3]$). On this augmented CSP, arc consistency will prune *all* the pairs that are inconsistent. The process of reformulating with the cross-domain representation and enforcing a local consistency can therefore be seen as enforcing a local super consistency. A super solution of the original CSP can be extracted by keeping only the first element of every pair.

Since the constraint graph of this CSP is a tree, successively enforcing arc-consistency and assigning a variable leads to a solution without backtracking. The possible solutions to this augmented CSP are: $\langle \langle 1,2 \rangle, \langle 2,1 \rangle, \langle 2,3 \rangle \rangle$, $\langle \langle 1,2 \rangle, \langle 2,1 \rangle, \langle 3,2 \rangle \rangle$, $\langle \langle 1,2 \rangle, \langle 2,3 \rangle, \langle 3,2 \rangle \rangle$ and $\langle \langle 2,1 \rangle, \langle 2,3 \rangle, \langle 3,2 \rangle \rangle$. The first corresponds to $\langle 1,2,2 \rangle$, the second and the third to $\langle 1,2,3 \rangle$ and the fourth to $\langle 2,2,3 \rangle$



5 Finding Super Solutions Via Search

5.1 Super Consistency

Local consistency allows backtrack-based search algorithms such as MAC to detect unsatisfiable subproblems earlier. Local consistency can also be used to develop efficient algorithms for finding super solutions. We shall introduce three ways of incorporating local consistency into a search algorithm for seeking super solutions. The first (AC+), a naive approach, augments the traditionnal arc-consistency by a further condition, achieving a very low level of filtering. The second (arc-consistency on $P \times P$) allows us to infer all possible local information, just as in arc-consistency in a regular CSP [4]. However this comes at a high computational cost, though polynomial. The third (super AC) approach gives less inference, but is a good tradeoff between the amount of pruning and complexity. Informally, a consistent closure of a CSP contains only partial solutions for a given level of locallity. However, the situation with super solutions is a little bit more tricky because values that do not get used in any local super solution can still be essential as a *repair* and thus cannot be simply pruned. Throughout the rest of this paper we will refer only to $(1, 0)$-super solutions, and thus to $(1, 0)$-super arc-consistency.

5.2 Arc-Consistency+

If S is a super solution, then for every variable, at least *two* values are consistent with all the others values of S. Consequently, being arc consistent *and having non-singleton domains* is a necessary condition of satisfiability. AC+ can then be defined as follows: for a CSP $P = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$: AC+$(P) \Leftrightarrow AC(P) \wedge \forall D \in \mathcal{D}, |D| > 1$. Whilst AC+ is usually too weak to give good results, it is the basis for an algorithm for the associated optimization problem (section 6).

5.3 Arc-Consistency on $P \times P$

$P \times P$ has the interesting property of having exactly the same topology as the original problem P. Moreover, each pair $\langle value, repair \rangle$ is explicitly represented, therefore, arc consistency on $P \times P$ makes all the possible inference, regarding arcs. The proof that AC on $P \times P$ is the tightest filtering introduced in this paper follows in section 5.5. As a corollary, tree and treewidth bounded tractable classes of CSPs are also tractable for finding $(1, 0)$-super solutions, through cross-domain

representation, since any tree structure is conserved by the transformation. In a similar way, if P is binary and Boolean, then $P+P$ is binary and Boolean, and hence tractable.

5.4 A Notion of Super Consistency

Arc Consistency on $P \times P$ allows us to infer all that can be inferred locally. In other words, we will prune any value in a cross-domain that is not locally consistent. However, this comes at high cost, maintaining arc-consistency will be $O(d^4)$ where d is the initial domain size. We therefore propose an alternative that does less inference, but at a much lower (for example, quadratic) cost. The main reason for the high cost is the size of the cross-domains. A cross-domain is quadratic in the size of the original domain since it explicitly represents the repair value for each super value. Here we will simulate (most of) the inference performed by super consistency, but will only look at one value at a time, and not pairs. We will divide the domain of the variable into two separate sets of domains: the "super domain" (SD) where only super values are represented and a "repair domain" (RD) where repair values are stored.

- A value v is in the super domain of X iff for any other variable Y, there exists v' in super domain of Y and r in repair domain of Y such that (v, v') and (v, r) are allowed and $(v' \neq r)$.
- A value v is in repair domain of X iff for any other variable Y, there exists v' in super domain of Y such that (v, v') is allowed.

The definition of super arc-consistency translates in a straightforward way into a filtering algorithm. The values are marked as either *super* or *repair*, and when looking for support of a super value, an additional and different support marked either as *super* or *repair* is required. The complexity of checking the consistency of an arc increases only by a factor of 2 and thus remains in $O(d^2)$. An algorithm that maintains such a consistency would branch only on the values in super domains and would fail if either the super domains becomes empty or the repair domains becomes singleton. The low cost of achieving super arc-consistency comes at the price of achieving a lower level of consistency compared to maintaining arc-consistency on $P \times P$, as shown in Figure 4. Moreover, the consistency must be maintained also on the domains of the variables already assigned. The super domain of an assigned variable is reduced to the chosen value and cannot change, but the repair domain can wipe out because of an assignment in the future. Figure 2 depicts an algorithm enforcing super AC on the CSP $X \leq Y \leq Z$. The first Figure shows the microstructure of the CSP. In the following Figure, super consitency is established: $Y = 1$ and $Y = 3$ have only one support, respectively on X and Z, thus they cannot be in super domain. Furthermore $X = 3$ and $Z = 1$ have each only one support on Y, respectively $Y = 1$ and $Y = 3$, which are not in the super domain. Hence $X = 3$ and $Z = 1$ are pruned. $Y = 2$ is the only value remaining in $SD(Y)$ and is thus assigned. In the last Figure, X is assigned to 2. As a result, $Y = 1$ is no longer supported and

is pruned. Since it was the second support for $Z = 2$ on Y, $Z = 2$ is no longer in the super domain anymore. $Z = 3$ is the only value remaining in $SD(Z)$ and is assigned.

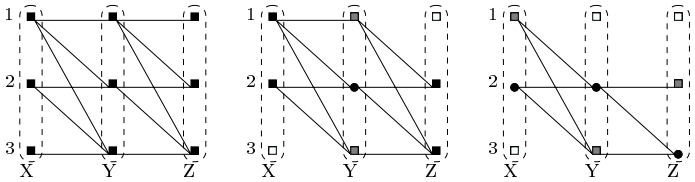


Fig. 2. Left: The microstructure of the CSP, Middle: its super consistent closure, Right: a super solution is reached after assigning $X = 2$.

- v is an assigned value ($SD = \{v\}$)
- v is in super domain ($v \in SD$) and in repair domain($v \in RD$)
- v is in repair domain ($v \in RD$)
- v is pruned

5.5 Theoretical Properties

For the theorem and the proof below, we use the notation (x)(P) to denote that the problem P is "consistent" for the filtering (x). We compare AC on $P \times P$, AC on $P+P$, AC+ and super AC.

Theorem 2 (level of filtering). *For any subproblem P, AC($P \times P$) $\Rightarrow$ super AC(P) $\Rightarrow$ AC($P+P$) $\Leftrightarrow$ AC+(P).*

Proof. **(1) AC($P+P$) $\Rightarrow$ AC+(P):** Suppose that P is not AC+, then in the arc consistent closure of P, there exists at least one domain D_i such that $|D_i| \leq 1$. $P+P$ contains P and then in its arc consistant closure, we have $|D_i| \leq 1$ as well. X_i is linked to a duplicate of itself which domain D'_i is then equal to D_i and therefore singleton (with the same value) or empty. However, recall that we force $X_i \neq X'_i$, thus $P+P$ is not AC.

(2) AC+(P) $\Rightarrow$ AC($P+P$): Suppose we have AC(P) and any domain D in P is such that $|D| > 1$, now consider $P+P$. The original constraints are AC since P is AC. The duplicated constraints are AC since they are identiqual to the original ones. The not_equal constraints between original and duplicated variables are AC since any variable has at least 2 values.

(3) super AC(P) $\Rightarrow$ AC+(P): Suppose that P is not AC+, then there exists two variables X, Y such that any value of X has at most one support on Y, therefore the corresponding super domain is wiped-out, and P is not super AC.

(4) super AC(P) $\not\Leftarrow$ AC+(P): See counterexample in Figure 3.

(5) AC($P \times P$) $\Rightarrow$ super AC(P): Suppose that AC($P \times P$), then for any two variables X, Y there exist two pairs $\langle v1, r1\rangle \in D(X) \times D(X), \langle v2, r2\rangle \in D(Y) \times D(Y)$, such that $\langle v1, r2\rangle, \langle r1, v2\rangle$ and $\langle v1, v2\rangle$ are allowed tuples. Therefore $v1$ belongs to the super domain of X and $v1$ and $r1$ belong to the repair domain

of X. Thus, the super domain of X is not empty and the repair domain of X is not singleton. Therefore, P is super AC.

(6) AC($P \times P$) $\not\Leftarrow$ super AC(P): See counterexample in Figure 4. □

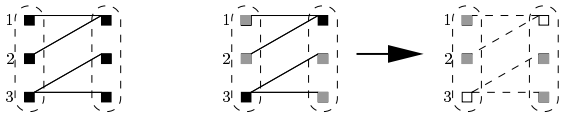


Fig. 3. The first graph shows the microstructure of a simple CSP, two variables and 3 values each, *allowed* combinations are linked. P is AC+ since the network is arc-consistent and every domain contains 3 values. However, P is not super AC since the greyed values (in the second graph) are not in super domains, they have only one support. In the second step, the whitened variables (in the third graph) are also removed from both repair and super domains since they don't have a support in a super domain.

Fig. 4. The first graph shows the microstructure of a simple CSP, three variables and four values each, *allowed* combinations are linked. P is super AC since the super domains are of size 2 (black values), and the repair domains are of size 4 (black and grey values). The second graph shows $P \times P$, which is not arc-consistent.

5.6 Super Search Algorithms

MAC+. This algorithm establishes AC+ at each node. That is, establishes AC and backtracks if a domain wipes out *or* becomes singleton. In the MAC algorithm, we only prune future variables, since the values assigned to past variables are guaranteed to have a support in each future variable. Here, this also holds, but the condition on the size of the domains may be violated for an assigned variable because of an assignment in the future. Therefore, firstly arc-consistency must be established on the whole network, and not only on the future variables. Secondly variables are not assigned in a regular way (which is usually equivalent to reducing its domain to the chosen value) but one value is marked as super value, that is added to the current partial solution, and unassigned values are kept in the domain: they are possible repairs so far. The algorithm can be informally described as follows:

- Choose a variable X
- Assign a value $v \in D(X)$ to X, but keep the unassigned values in $D(X)$
- For all $Y \neq X$, backtrack if Y has not at least two supports for v
- Revise the constraints as the MAC algorithm, and backtrack if the size of any domain falls bellow 2.

Super MAC. We give the pseudo code of `super MAC` in Figure 5. This algorithm is very similar to MAC algorithm, the super domains ($\mathcal{SD}$) and repair domains ($\mathcal{RD}$), are both equal to the original domains for the first call. Most of the differences are grouped in the procedure `revise-Dom`. The values are pruned according to the rules described in section 5.4 (loop 1), and the algorithm backtracks if a super domain wipes out or a repair domain becomes singleton (line 2). Note that, as for MAC+, the consistency is also established on the domains of the assigned variables, (`super AC` loop 1).

We have established an ordering relation on the different filterings. However, for the two algorithms above, the process of assigning a value to a variable in the current solution doesn't lead to the same subproblem as in a regular algorithm. Whereas for a regular backtrack algorithm, the domains of the assigned variables are reduced to the chosen value, some unassigned values are still in their domain (but marked only as "repair") for the algorithms above. We have proved that a problem P is AC+ iff P+P is AC. However, consider the subproblem P' induced by the assignment of X by MAC+. P'+P' may have more than one variable in X, whereas the corresponding assignment in P+P leaves only one value in the domain of X (see Figure 6). Therefore the ordering on the consistencies doesn't hold for the number of backtracks of the algorithms themselves. However, MAC($P\times P$) never backtracks when any other algorithm doesn't, and MAC+ always backtracks when any other algorithm does. Therefore any solution found by MAC($P\times P$) will eventually be found by other algorithms, and MAC+ will find any solution found by another algorithm. We prove that MAC+ is correct and MAC($P\times P$) is complete. Hence all four algorithms are correct and complete.

Theorem 3. *For any given CSP P, the sets of solutions of MAC+(P), of super MAC(P), of MAC($P\times P$), and of MAC(P+P) are the same and is the set of all super solutions to P.*

Proof. MAC+ is correct: suppose that S is not a super solution, then there exists a variable X assigned to v in S, such that $\forall w \in D(X), v \neq w$, w cannot replace v in S. Therefore when all the variables are assigned, and thus, remain in the domains only the values that are arc-consistent with the current solution, $D(X) = \{v\}$ and then S is not returned by MAC+.
MAC($P\times P$) is complete: let S be a super solution, for any variables X, Y, let $v1$ be the value assigned to X in S, and $r1$ one of its possible repairs. Similarly $v2$ is the value assigned to Y and $r2$ its repair. It's easy to see that the pairs $\langle v1, r1\rangle$ and $\langle v2, r2\rangle$ are super arc-consistent, i.e, $\langle v1, v2\rangle$, $\langle v1, r2\rangle$ and $\langle r1, v2\rangle$ are allowed tuples. □

6 Finding the Most Robust Solutions

Finding super solutions may be difficult because (1) from a theoretical perspective, the existence of a backbone variable guarantees the non-existence of super solutions, and (2) from an experimental perspective (see next section), it is quite rare, even if we have no backbone variables, to have super solutions

Algorithm 1: `super MAC`

Data : CSP: $P = \{\mathcal{X}, \mathcal{SD}, \mathcal{RD}, \mathcal{C}\}$, solution: $S = \emptyset$, variables: $\mathcal{V} = \mathcal{X}$
Result : Boolean // $\exists S$ a $(1,0)$-super solution
if $\mathcal{V} = \emptyset$ **then** return True;
choose $X_i \in \mathcal{V}$;
foreach $v_i \in SD_i$ **do**
  save $\mathcal{SD}$ and $\mathcal{RD}$;
  $SD_i \leftarrow \{v_i\}$;
  if `super AC`$(P, \{X_i\})$ **then**
    if `super MAC`$(P, S \cup \{v_i\}, \mathcal{V} - \{X_i\})$ **then** return True;
  end
  restore $\mathcal{SD}$ and $\mathcal{RD}$;
end
return False;

Algorithm 2: `super AC`

Data : CSP: $P = \{\mathcal{X}, \mathcal{SD}, \mathcal{RD}, \mathcal{C}\}$, Stack: $\{X_i\}$
Result : Boolean // P is super arc consistent
while *Stack is not empty* **do**
  pop X_i from Stack;
1 **foreach** $C_{ij} \in \mathcal{C}$ **do**
    switch `revise-Dom`(SD_j, RD_j, SD_i, RD_i) **do**
      case *not-cons*
        return False;
      case *pruned*
        push X_j on Stack;
    endsw
  end
end
return True;

Procedure `revise-Dom`(SD_j, RD_j, SD_i, RD_i) : {pruned,not-cons,nop}

1 **foreach** $v_j \in SD_j$ **do**
  if $\nexists v_i \in SD_i, v_i' \in RD_i$ *such that* $\langle v_i, v_j\rangle \in C_{ij} \wedge \langle v_i', v_j\rangle \in C_{ij} \wedge v_i \neq v_i'$ **then**
    $SD_j \leftarrow SD_j - \{v_j\}$;
  end
end
foreach $v_j \in RD_j$ **do**
  if $\nexists v_i \in SD_i$ *such that* $\langle v_i, v_j\rangle \in C_{ij}$ **then**
    $RD_j \leftarrow RD_j - \{v_j\}$;
  end
end
if *at least one value has been pruned* **then** return pruned;
2 **if** $|SD_j| = 0 \vee |RD_j| < 2$ **then** return not-cons;
return nop;

Fig. 5. super MAC algorithm

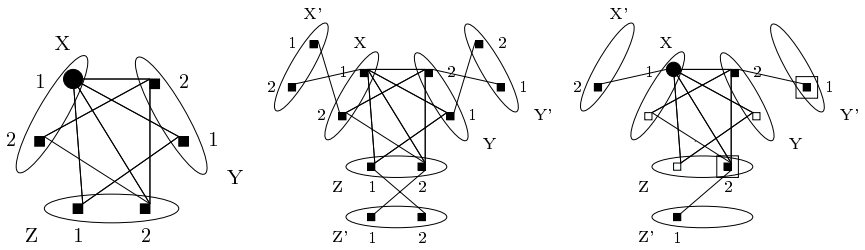


Fig. 6. Left: A CSP P, P is still AC+ after assigning X to 1. Middle: P+P, each variable has a duplicate which must be different, the constraints linking those variables are not represented here, the constraints on X' are exactly the same as the ones on X. Right: When the same assignment, $X = 1$ is done in P+P, we have the following propagation $X' \neq 1 \rightarrow Y \neq 1 \wedge Z \neq 1 \rightarrow Y' \neq 2 \wedge Z' \neq 2$. Now consider $(Y' : 1)$ and $(Z : 2)$ they are not allowed, the network is no longer arc-consistent.

where all variables can be repaired. To cure both problems, we propose finding the "most robust" solution that is as close as possible to a super solution. An optimal solution is defined as a solution that is as close as possible to a super solution.

For a given solution S, a variable is said to be repairable iff there exist at least a value in its domain different from the one assigned in S, and compatible with all other values in S.

The optimal solution is a solution where the number of repairable variables is maximal. Such an optimal robust solution is guaranteed to exist. If the problem is satisfiable, we will have a solution where, in the worst case, none of the variables are repairable. We hope, of course, to find some of the variables are repairable. For example, our experiments show that satisfiable instances at the phase transition and beyond have a core of roughly $n/5$ repairable variables. To find such solutions, we propose a branch and bound algorithm that finds an optimal robust solution. The algorithm implemented is very similar to MAC+ (see 5.6), where arc-consistency is established on the non-assigned as well as on the assigned variables. The current lower bound computed by the algorithm is the number of singleton domains, the initial upper bound is n. Indeed, each singleton domain corresponds to an *unrepairable* variable, since no other value is consistent with the rest of the solution. The rest is a typical branch and bound algorithm. The first solution (or the proof of unsatisfiability) needs exactly the same time as the underlying MAC algorithm. Afterwards, it will continue branching and discovering better solutions. It can therefore be considered as an *incremental anytime algorithm*. We refer to it as super B&B.

For optimization problems, the optimal solution may not be a super solution. We can look for either the most repairable optimal solution or the super solution with the best value for the objective function. More generally, an optimization problem then becomes a *multi-criterion* optimization problem, where we are optimizing the number of repairable variables and the objective function.

7 Experimental Results

Due to the lack of space, this section gives only some observations from our experiments, the interested reader is pointed to the technical report [2]. Our aims were firstly to study the difficulty of finding super solutions, rather than solutions. As expected, for a given density, the phase transition for MAC begins at a tightness much larger than the end of the phase transition for super MAC. That means that a hard problem is very likely not to have a super solution. Moreover, the highest point of the phase transition peak is orders of magnitude greater for finding super solutions than for finding solutions. However, we found that, on the bandwidth problem, for instance, by slightly relaxing the problem (in that case the optimality criterion) super solutions can be found, or alternatively, optimal non-super solutions with a relatively high number of repairable variables.

We present here the comparison between MAC on $P \times P$, super MAC, MAC+ and MAC on $P+P$, on two classes of random problem instances at the phase transition: (50 variables, 15 values, 100 constraints, 114 disallowed tuples per constraint) and (100 variables, 6 values, 250 constraints, 10 disallowed tuples per constraint). Figure 7 gives the cpu time, and the number of backtracks of these algorithms. We observe that (1) MAC on $P \times P$ effectively prunes more than all other methods, but is not practical when the domain size is too big, and (2) super MAC outperforms all other algorithms, in terms of runtime, as soon as the size of the problem increases.

	MAC+	MAC on $P+P$	MAC on $P \times P$	super MAC
$\langle n = 50\ d = 15\ p_1 = 0.08\ p_2 = 0.5 \rangle$				
CPU time (s)	788	43	53	**1.8**
backtracks	152601000	111836	**192**	2047
time out (3000 s)	12%	**0%**	**0%**	**0%**
$\langle n = 100\ d = 6\ p_1 = 0.05\ p_2 = 0.27 \rangle *$				
CPU time (s)	2257	430	3.5	**1.2**
backtracks	173134000	3786860	**619**	6487
time out (3000 s)	66%	7%	**0%**	**0%**

Fig. 7. Results at the phase transition. $*$ only 50 instances of this class were given to MAC+

8 Conclusion

Summary. We have studied the properties of supermodels within a CSP framework. We introduced the notion of super consistency, and based upon it, a search algorithm, super MAC is developed to solve the problem, which outperforms the other methods studied here. We also propose super B&B as an optimzation algorithm which finds the most robust solution that is as close as possible to a super solution.

Related work. Supermodels [8] and fault tolerant solutions [12] have been discussed earlier. Neighborhood interchangeability [6] and substitutability are closely related to our work, but whereas, for a given problem, interchangeability is a property of the values and works for all solutions, reparability is a property of the values according to a certain solution (it can be seen either as a property of a variable, or of the value given to this variable into a solution). Therefore those properties are incomparable. For instance, by definition if no solutions exists, then any two values are interchangeable whilst there is no repairable variable. On the other hand consider a CSP with two variables X, Y and the following solutions: $\langle 1,1\rangle, \langle 1,2\rangle, \langle 2,1\rangle, \langle 2,3\rangle$. This problem has two super solutions, $\langle 1,1\rangle$ and $\langle 2,1\rangle$. The value $X = 1$ is repairable in both super solutions by $X = 2$, and vice versa, but neither $X = 2$ is substitutable to $X = 1$, since $\langle 1,2\rangle$ is solution and not $\langle 2,2\rangle$, nor $X = 1$ is substitutable to $X = 2$, since $\langle 2,3\rangle$ is solution and $\langle 1,3\rangle$ is not.

Future directions. The problem of seeking super solutions becomes harder when multiples repairs are allowed, i.e, for $(1, b)$-super solutions. We aim to generalize the idea of super consistency to $(1, b)$-super solutions. In a similar direction, we would like to explore tractable classes of $(1, b)$-super CSPs. Furthermore, as with dynamic CSPs, we wish to consider the loss of n-ary no-goods and not just unary no-goods.

References

1. A. Dechter and R. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings AAAI-88*, pages 37–42, 1988.
2. B. Hnich E. Hebrard and T. Walsh. Super csps. Technical Report APES-66-2003, APES Research Group, 2003.
3. D. W. Fowler and K. N. Brown. Branching constraint satisfaction problems for solutions robust under likely changes. In *Proceedings CP-00*, pages 500–504, 2000.
4. E. C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32:755–761, 1985.
5. E. C. Freuder. Partial Constraint Satisfaction. In *Proceedings IJCAI-89*, pages 278–283, 1989.
6. E. C. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proceedings AAAI-91*, pages 227–233, 1991.
7. N. Jussien, R. Debruyne, and P. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Proceedings CP-00*, pages 249–261, 2000.
8. A. Parkes M. Ginsberg and A. Roy. Supermodels and robustness. In *Proceedings AAAI-98*, pages 334–339, 1998.
9. I. Miguel. *Dynamic Flexible Constraint Satisfaction and Its Application to AI Planning*. PhD thesis, University of Edinburgh, 2001.
10. T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problem. *IJAIT*, 3(2):187–207, 1994.
11. T. Walsh. Stochastic constraint programming. In *Proceedings ECAI-02*, 2002.
12. R. Weigel and C. Bliek. On reformulation of constraint satisfaction problems. In *Proceedings ECAI-98*, pages 254–258, 1998.

Steps toward Computing Flexible Schedules

Nicola Policella[1*,**], Stephen F. Smith[2], Amedeo Cesta[1], and Angelo Oddi[1]

[1] Planning & Scheduling Team, Institute for Cognitive Science and Technology
Italian National Research Council
Rome, Italy
{policella,cesta,oddi}@ip.rm.cnr.it
[2] The Robotics Institute, Carnegie Mellon University
Pittsburgh, PA, USA
sfs@cs.cmu.edu

Abstract. In this paper we consider the problem of building schedules that retain temporal flexibility. Such a feature represents a relevant benefit for managing changes in a dynamic environment. We begin by formalizing the concept of flexibility, to provide a set of metrics against which the flexibility of competing schedules can be compared. Then, using a common solving framework, we develop two orthogonal procedures for constructing a flexible schedule. The first, which we call the resource envelope based approach, uses computed bounds on cumulative resource usage (i.e., a resource envelope) to identify potential resource conflicts, and progressively winnows the total set of temporally feasible solutions into a smaller set of resource feasible solutions by resolving detected conflicts. The second, referred to as the earliest start time approach, instead uses conflict analysis of a specific (i.e., earliest start time) solution to generate an initial fixed-time schedule, and then generalizes this solution to a set of resource feasible solutions. We evaluate the relative effectiveness of these two procedures on a set of project scheduling benchmark problems, considering both their problem solving performance and the flexibility of the solutions they generate.

1 Introduction

In most practical scheduling environments, off-line schedules can have a very limited lifetime and scheduling is really an ongoing process of responding to unexpected and evolving circumstances. In such environments, insurance of robust response is generally the first concern. Unfortunately, the lack of guidance that might be provided by a schedule often leads to myopic, sub-optimal decision-making.

In this paper we pursue the idea of promoting robust response through the generation of flexible schedules – schedules that encapsulate a set of possible execution futures and hence can accommodate some amount of executional uncertainty. Our particular focus is generation of schedules that retain temporal

* Ph.D. student at the Department of Computer and Systems Science, University of Rome "La Sapienza", Italy
** Visiting student scholar at the Robotics Institute, Carnegie Mellon University

flexibility. Historically, a major obstacle to generating temporally flexible schedules has been the difficulty of accurately computing the number of resources required across all possible executions. Without this capability, it is difficult to obtain sufficient search guidance to achieve scalable problem solving performance. In [1], this problem is circumvented through use of a two-step procedure, where attention is first focused on generating a particular resource-feasible solution, the earliest start time solution, and then this solution is generalized into a flexible solution. However, in [6], a new procedure for computing resource usage bounds for a flexible schedule has been proposed, referred to as the *resource envelope*. Since this procedure generates "the tightest possible resource-level bound for a flexible plan", it suggests the possibility of generating a flexible schedule in a more direct, least-commitment fashion, using the resource envelope to detect potential resource conflicts and transforming the set of possible solutions into a small set of resource-feasible solutions by successively posting new conflict-resolving constraints between competing activities. Intuitively, we might expect that a schedule generation scheme which operates in such a least-commitment fashion would produce a solution with greater flexibility than an approach that instead produces a single point solution and attempts to generalize from this. But the performance tradeoffs are not immediately clear. To investigate these tradeoffs, we develop concrete implementations of each of the above approaches. To sharpen the comparison, we utilize a common scheduling framework wherein schedule generation is formulated as an incremental, conflict removal (or leveling) process. We experimentally compare the performance of each procedure on a set of known resource-constrained project scheduling problems, considering both problem solving and solution flexibility characteristics.

The paper starts by discussing the concept and benefits of flexible solutions in uncertain environments and specifying two parameters for measuring solution quality along this dimension (Sect. 2.1). In Sect. 3 we describe the precedence constraint posting (PCP) framework in which the two schedule generation approaches are to be defined and compared. In Sect. 4 and Sect. 5 the resource envelope and the earliest start time approach are respectively introduced. Section 5.1 then presents a method for obtaining flexible solutions from fixed time ones. An empirical evaluation is presented in Sect. 6, analyzing the feature of flexibility in Sect. 6.1. Finally we summarize our main results.

2 Flexibility and the Uncertainty in Scheduling

In the realm of scheduling problems different sources of uncertainty can arise: durations may not be known, resources may have lower capacity than expexted (i.e., machine breakdown), new tasks may need to be taken into account. Given this, one highly desirable characteristic of a schedule is that the reactions to unexpected events during execution entail small and localized changes.

One way to face this problem consists of using on-line (reactive) approaches. These approaches try to repair the schedule each time a new disruption happens. Keeping the pace with execution requires that the repair process be both fast and complete. A repair must be fast because of the need to re-start execution of the schedule as soon as possible. A repair also has to be complete in the sense that it has to take into account all changes that have occurred, avoiding to produce

new ones. As these two goals can be conflicting a compromise solution is often needed. Different approaches exist and they tend to favor either the swiftness of their reaction [10] or the completeness of the new solution [8].

Alternative approaches to managing execution in dynamic environments have focused on building schedules that retain flexibility and hedge against uncertainty (off-line or proactive approaches). Robust approaches aim at building solutions able to absorb some level of unexpected event without rescheduling. To achieve such a feature, different techniques have been investigated. One consists of building redundancy-based solutions, both of resources and of time, taking into account the uncertainty present in the domain [3]. An alternative technique is to construct a set of contingencies (i.e., a set of different solutions) and use the most suitable with respect to the actual evolution of the environment [4]. An important point to note is that both types of approaches above need to be aware of the possible events that can occur in the environment. In some cases, this need for knowledge about the uncertainty in the operating environment can present a barrier to their use.

For this reason, in the perspective of robust approaches, we consider a less knowledge-intensive approach: to simply build solutions that retain temporal flexibility where problem constraints allow. The aim is to produce solutions that enable reaction to exogenous events without *large changes* or *explicit assistance* (or repairs). A similar concept of producing solutions that promote bounded, localized recovery from execution failures is also proposed in [5]. The two conditions above are desired to insure an ability to keep pace with execution and, at the same time, maintain stability in the solution. To achieve these features, the idea is to construct partially ordered solutions, by introducing ordering constraints to resolve resource conflicts between pairs of activities. By providing greater execution flexibility, such solutions are more advantageous than fixed-time schedules (where precise start and end times are assigned to all activities). Fixed-time schedules are quite brittle and it is typically very difficult to follow them exactly during execution. Moreover, a flexible solution allows explicit reasoning about the uncontrollability of external events and the ability to include execution countermeasures.

2.1 Evaluation criteria

A fundamental point related to the flexibility concept introduced above is the need for metrics that characterize the quality of a flexible solution, and in general, the extent to which a solution is suitable for the execution phase. Different concepts can be used to describe the behavior of a given system facing uncertainty in the world - stability, flexibility or robustness - but these notions remain vague unless both the perturbations and features of interest are specified. Indeed it makes no sense to define the quality of a system without first specifying which of its characteristics have been considered. In the following we represent the scheduling problem by a graph where for each activity a_i there are two nodes (events), the start time s_{a_i} and the end time e_{a_i} and for each constraint there is an edge in the graph. Applying Dijkstra's Shortest Path Algorithm the earliest and latest values for both events of each activity are computed: $est(a_i)$, $lst(a_i)$ $eet(a_i)$ and $let(a_i)$.

In [1] a metric[3] based on the temporal slack associated with each activity is introduced:

$$flex = \sum_{h \neq l} \frac{|d(e_{a_h}, s_{a_l}) - d(s_{a_l}, e_{a_h})|}{H \times N \times (N-1)} \times 100 \quad (1)$$

in which H is the horizon of the problem, N is the number of activities and $d(tp_1, tp_2)$ is the distance between the two time points. This metric aims at measuring the *fluidity* of a solution, i.e., the ability to use flexibility to absorb temporal variation in the execution of activities. The higher the value of $flex$, the less the risk of a "domino effect", i.e. the higher the probability of localized changes.

While the previous parameter measures the ability to avoid domino effects, another aspect of solution flexibility is the expected magnitude of potential changes. We introduce a new parameter that takes into account the impact of disruptions on the schedule, or *disruptibility* of a solution:

$$dsrp = \frac{1}{N} \sum_{i=1}^{N} Pr_{disr}(a_i) \times \frac{let(a_i) - eet(a_i)}{num_{changes}(a_i, \Delta_{a_1})} \quad (2)$$

where $Pr_{disr}(a_i)$ is the probability that a disruption occurs during the execution of the activity a_i. The value $let(a_i) - eet(a_i)$ represents the temporal flexibility of each activity a_i, i.e., the ability to absorb a change in the execution phase. The probability is considered because the flexibility of each activity gives a different contribution to the solution quality according to the possibility that a disruption can occur, or not, during its execution. Through the function $num_{changes}(a_i, \Delta_{a_i})$ the number of entailed changes given a right shift Δ_{a_i} of the activity a_i is computed. In Sect. 6.1 both the probability distribution, $Pr_{disr}(a_i)$, and the right shift, Δ_{a_i}, used in the empirical evaluation are described.

The intuition behind this parameter consists of considering the trade-off between the flexibility of each activity, $let(a_i) - eet(a_i)$, and the number of changes implied, $num_{changes}(a_i, \Delta_{a_i})$. The latter can be seen as the price to pay for the flexibility of each activity.

3 A Precedence Constraint Posting Framework

The goal of the paper is to evaluate the ability to find flexible solutions using two different methods to estimate the resource needs at each instant: the earliest start time profile [1] and the resource envelope [6]. For performing such a comparison we will use each to guide the search within a profile-based scheduling framework. Within this framework, a resource feasible solution is produced by progressively detecting time periods where resource demand is higher than resource capacity and posting sequencing constraints between competing activities to reduce demand and eliminate capacity conflicts. There are different ways of representing and maintaining profile information. We will compare the extreme ones: the resource envelope, which maintains all possible solutions, and the earliest start time approach, which considers a single solution. In the latter case, as

[3] Named *rb*, robustness, in [1].

PCP-greedy($Problem$)
Input: A problem
Output: A conflict-free solution
1. $CurrentSituation \leftarrow Problem$
2. **if** Exists-Unresolvable-Conflict($CurrentSituation$)
3. **return** NIL
4. **else**
5. $ConflictSet \leftarrow$ Select-Conflict-Set($CurrentSituation$)
6. if $ConflictSet = \emptyset$
7. **return** $CurrentSituation$
8. **else**
9. $constraint \leftarrow$Select-Leveling-Constraint($ConflictSet$)
10. Add-Constraint($CurrentSituation$, $constraint$)
11. PCP-greedy($CurrentSituation$)

Fig. 1. Conflict-free Algorithm

it finds a single solution, a robust solution will be built as a two step process of finding a fixed-time solution and then generalizing from it as current constraints will permit.

The framework is based on a constraint satisfaction model of scheduling problems in which each activity a_i is represented by two events, the start time s_{a_i} and the end time e_{a_i}. There are two aspects to take into account: the time and the resource constraints. The former introduces a set of constraints that represent either the duration of the activity, $dur_{a_i}^{min} \leq e_{a_i} - s_{a_i} \leq dur_{a_i}^{max}$, or the relation between a pair of activities, $c_{ij}^{min} \leq s_{a_j} - s_{a_i} \leq c_{ij}^{max}$. Representing the resources requires taking into account the usage of each resource r_k by the different activities. This is done by associating a resource usage value at each event, $ru_{ik}(tp)$. This allows the representation of different kind of activities: for instance, for an activity a_i that uses ru_{ik} capacity units it will be sufficient to set $ru_{ik}(st_{a_i}) = ru_{ik}$ and $ru_{ik}(et_{a_i}) = -ru_{ik}$. According to previous models the resource constraint for a resource r_k is defined as $\sum_{\forall tp_j \leq t} ru_{ik}(tp_j) \leq cap_k$ for each instant t.

Fig. 1 shows the conflict removal procedure. Given a problem, in terms of a partial ordered plan, the first step consists of building an estimate of the resource levels needed (lines 2–5). This analysis can highlight an infeasible current situation, where resource needs are greater then the availability: *contention peak* (line 6). For solving such a case, a new precedence constraint is synthesized and added to the problem (lines 9-11). What is needed to configure a complete search procedure are mechanisms and heuristics for recognizing, prioritizing and resolving conflicts. These strategies derive from those first introduced in [11] and extended to the cumulative resource case in [1]. A conflict is defined to be any pair $\langle a_i, a_j \rangle$ of activities in a given contention peak. Four possible conditions can held between the two activities according to the maximum distance, $d()$, between two events:

condition 1 : $d(e_{a_i}, s_{a_j}) < 0 \wedge d(e_{a_j}, s_{a_i}) < 0$. In this case there is no way to order the activities. This is identified as a *pairwise unresolvable conflict*.

condition 2 : $d(e_{a_i}, s_{a_j}) < 0 \wedge d(e_{a_j}, s_{a_i}) \geq 0 \wedge d(s_{a_i}, e_{a_j}) > 0$. There is only one feasible ordering the two activities $a_j\{before\}a_i$.

condition 3 : $d(e_{a_i}, s_{a_j}) \geq 0 \wedge d(e_{a_j}, s_{a_i}) < 0 \wedge d(s_{a_j}, e_{a_i}) > 0$. Like the previous one this is also a *pairwise uniquely resolvable conflict*. In this case the relation is $a_i\{before\}a_j$.

condition 4 : $d(e_{a_i}, s_{a_j}) \geq 0 \wedge d(e_{a_j}, s_{a_i}) \geq 0$. In this case we have a *pairwise resolvable conflict*. Both orderings $a_i\{before\}a_j$ and $a_j\{before\}a_i$ are feasible and a choice is needed.

The previous conditions are used for implementing the following functions utilized in the general schema introduced in Fig. 1:

Exists-Unresolvable-Conflict(*CurrentSituation*). This procedure identifies whether the current situation is infeasible, by propagating the constraints defined in the problem. It detects a contention peak where for each pair of activities condition 1 holds.

Select-Conflict-Set(*CurrentSituation*). This procedure selects a pair $\langle a_i, a_j \rangle$ of activities within a resolvable peak. Two cases are distinguished. When one or more pairwise conflicts satisfy conditions 2 or 3 then the conflict with the minimum (and negative) value $\omega_{res}(a_i, a_j) = \min\{d(e_{a_i}, s_{a_j}), d(e_{a_j}, s_{a_i})\}$ is selected. Alternatively, if condition 4 holds, then is selected the pairwise conflict $\langle a_i, a_j \rangle$ that minimize the value

$$\omega_{res}(a_i, a_j) = \min\{\frac{d(e_{a_i}, s_{a_j})}{\sqrt{S}}, \frac{d(e_{a_j}, s_{a_i})}{\sqrt{S}}\}$$

where $S = \frac{\min\{d(e_{a_i}, s_{a_j}), d(e_{a_j}, s_{a_i})\}}{\max\{d(e_{a_i}, s_{a_j}), d(e_{a_j}, s_{a_i})\}}$.

Select-Leveling-Constraint(*ConflictSet*). This procedure returns the ordering constraint that leaves the most temporal flexibility: $a_i \leq a_j$ whether $d(e_{a_i}, s_{a_j}) > d(e_{a_j}, s_{a_i})$ and $a_i \geq a_j$ otherwise.

As the reader can see, decisions are taken according to a least commitment principle, trying to retain the maximum amount of temporal flexibility. For that reason the values of the distances $d(e_{a_i}, s_{a_j})$ and $d(e_{a_j}, s_{a_i})$ have a key role.

To explore the impact of additional heuristic bias on the effectiveness of various instantiations of this greedy search algorithm, we also define an enhanced Select-Conflict-Set procedure which incorporates a further heursitic estimator. Specifically, we add a method for analyzing conflict sets with the aim of avoiding redundant constraints, through identification of *Minimal Critical Sets* [7]. A *Minimal Critical Set*, MCS, is *a set of activities that simultaneously requires a resource r_j with a combined capacity requirement greater than its capacity c_i and the requirement of any subset is lower than, or equals to c_i*. Application of this method can be seen genrally as a filtering step. It extracts from several conflict sets those sub-sets of activities that are necessary to solve. In [7] a heuristic estimator is also provided. Given a MCS and a set of possible ordering constraints $\{oc_1, \ldots, oc_k\}$ which can be posted between pairs of the activities in MCS the estimator $K(\text{MCS})$ is defined:

$$\frac{1}{K(\text{MCS})} = \sum_{i=1}^{k} \frac{1}{1 + commit(oc_i) - commit(oc_{min})} \tag{3}$$

where $commit(oc_i)$ estimates the loss in temporal flexibility as explained in [7]. As a matter of fact the high computational complexity of enumerating all MCSs prohibits its use on scheduling problems of any interesting size. In [2] two methods to overcome such a problem are described. They consist of sampling a subset of the set of all MCSs.

Linear sampling. A queue Q is used to select an MCS from a contention peak P. Activities a_i are considered sequentially and inserted in Q until the sum of the resource requirement is greater than the resource availability. Then the set Q is saved in a list of MCS and the first element in Q is removed. The previous steps are iterated until there are no more activities.

Quadratic sampling. This is an extension of the previous schema in which the second step is expanded as follows. Once the correct MCS has been collected, instead of removing the first element from Q a forward search through the remaining activities is performed to collect all MCS that can be obtained by dropping the last item placed in Q and substituting with single subsequent activities until an MCS is composed.

This heuristic estimator leads to a modified Select-Conflict-Set procedure (line 5 of the algorithm in Figure 1): it chooses the MCS with highest K value. The conflict resolution heuristic (Select-Leveling-Constraint) simply chooses oc_{min}.

The next sections introduce the two approaches of interest in this paper for representing and maintaining resource profile information, the resource envelope approach and the earliest start time approach.

4 The Resource Envelope

The first method considered for guiding the search for reaching flexible solutions is the resource envelope defined in [6]. This work proved that it is possible to find "the tightest possible resource-level bound for a flexible plan" through a polynomial algorithm. The advantage of using the resource envelope is that all possible temporal allocations are taken into account during the solving process. Thus, unlike the fixed time approaches, a solution consists of a set of feasible solutions. In the remainder of this section we briefly review the idea behind the computation of the resource envelope.

To find the maximum (minimum) value of the resource level in an instant t most methods subdivide the set of time points (events) into the following subsets:

- B_t: the set of events tp_i s.t. $let(tp_i) \leq t$;
- E_t: the set of events tp_i s.t. $est(tp_i) \leq t < let(tp_i)$;
- A_t: the set of events tp_i s.t. $est(tp_i) > t$.

Since the events in B_t are those which will end before or at time t, they all contribute, with the associated resource usage $ru_{ik}(tp_i)$, to the value of the resource profile of r_k in the instant t. By the same argument we can exclude from such a computation the events in A_t. Then the crucial point is to determine which of those in E_t have to be take into account. A basic method consists of enumerating all the possible combinations of events in E_t. This method implies a high computational cost, and for this reason approximate techniques have been

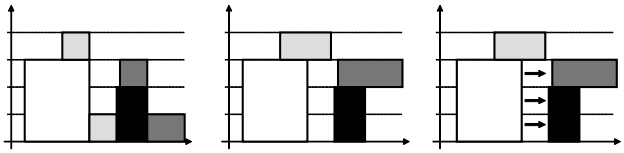


Fig. 2. Chaining method: intuition

developed. In [6], instead, the author proves that to find the subset of E_t for computing the upper (lower) bound, it is possible to avoid such an enumeration. He shows that a polynomial algorithm can be found, taking the relations among the events into account through a reduction to a well-known tractable problem: the Max-Flow Problem. The effectiveness of the reduction is due to the fact that it allows to underline the relations among the set of the events and to consider the subset of feasible combinations. The details of the algorithm are omitted here. We simply recall that the method broadly consists of building a Max-Flow problem from the set of events belonging to E_t and, after the max flow is found, the subset $P_{max} \subseteq E_t$ (P_{min}), of events that gives the maximum (minimum) value of the resource level at the instant t, is computed by collecting all activities that are reachable from the source in the residual graph of the Max-Flow problem. We will discuss the approach obtained using the resource envelope to guide the search (EBA) in Sect. 6.

5 The Earliest Start Time Approach

In [1] it has been shown that use of the earliest start time profile is an effective way to solve scheduling problems. This profile is based on a temporal net property: at each time point (event) tp_i there is an associated interval of possible values $[lb_{tp_i}, ub_{tp_i}]$ and the extremes of the interval if chosen as the value for all time variables tp_i identify a solution of the temporal net. In [1] the earliest start time solution (that is $tp_i = lb_{tp_i}$ for each i) is considered. The method (named ESTA) consists of building the resource profile for such a temporal solution and matching it with the resource bounds. If a violation exists then further constraints are posted to resolve the resource conflict.

The fundamental difference between the earliest start time approach with respect to the resource envelope approach is that while the latter gives a measure of the worst-case hypothesis, the former identifies "real" problems/conflicts in a particular situation (earliest start time solution). In other words the first approach says what *can* happen in such a situation relative to the entire set of possible solutions, the second one, instead, what *will* happen in such a particular case.

As ESTA finds fixed time solutions a method for computing flexible solutions is needed. In the next section we describe a method for achieving such a feature.

Chaining(*Problem, Fixed-Time Solution*)
Input: A problem and an its Fixed-Time solution
Output: A flexible solution
1. $S \leftarrow$ *Fixed-Time Solution*
2. $S^* \leftarrow$ *Problem*
3. Sort all the activities according to their start time in S
4. **For each** activity a_i
5. **For each** resource r_j
6. k=1
7. **While** $ru_{ij} > 0$
8. **If** $Q_{jk} \neq \emptyset$
9. $(a, t_1, t_2) \leftarrow ReadLastElement(Q_{jk})$
10. **If** $est_{a_i} \geq t_2$
11. Add $(a_i, est_{a_i}, eet_{a_i})$ to Q_{jk}
12. Add-Constraint(S^*, $a\{before\}a_i$)
13. $ru_{ij} = ru_{ij} - 1$
14. **else**
15. Add $(a_i, est_{a_i}, eet_{a_i})$ to Q_{jk}
16. $ru_{ij} = ru_{ij} - 1$
17. $k = k + 1$
18. **Return** S^*

Fig. 3. Chaining Algorithm

5.1 Producing flexible solutions

In [1] the authors suggest an approach for translating a fixed schedule to a MCM-SP problem instance into a flexible solution. The MCM-SP problem involves a set of activities a_i, each of them requiring only the use of a single resource for its entire duration. Given a solution the transforming method, named *chaining*, consists of creating sets of chains of activities, one set for each resource. This operation is accomplished by deleting all previously posted leveling constraints and using the solution resource profiles to post a new set of constraints. In this section we generalize that method for problems that involve multi-capacited resources. This requires a few adjustments:

- a first step is to consider a resource r_j with capacity c_j as a set R_j of $n = c_j$ single capacity sub-resources. The idea is to create a similar situation to the MCM-SP case;
- in this light the second step is to ensure that each activity is allocated to the same subset of R_j. This step can be viewed in Figure 2: on the left there is the resource profile of a resource r_j, each activity is represented with a different color. The second step consists of maintaining the same subset of sub-resources for each activity over time. For instance, in the center of Figure 2 the light gray activities is re-drawn in the way that it is always allocated on the fourth sub-resource;
- the last step is to build a chain for each sub resource in R_j. On the right of Figure 2 this step is represented by the added constraints. That explains why the second step is needed. Indeed if the chain is built taking into account only the resource profile, there can be a problem with the relation between

the light gray activity and the white one. In fact, using the chain building procedure just described, one should add a constraint between them, but that will not be sound. The second step allows, indeed, to avoid this problem, taking into account the different allocation on the set of sub-resources R_j.

Figure 3 contains the sketch of a chaining algorithm. It uses a set of queues, Q_{jk}, to represent each capacity unit of the resource r_j. The elements of the queues consists of a triple $\langle a_i, est_{a_i}, eet_{a_i} \rangle$, that is, the activity a_i and its start and end time according to the earliest start time solution S. The algorithm starts by sorting the set of activities according to their start time in the solution S. Then it proceeds to allocate the capacity units needed for each activity. It selects only the capacity units available at the start time of the activity (line 10). Then when an activity is allocated on a queue a new constraint between this activity and the previous in the queue is posted (line 12).

The enhanced algorithm obtained by adding the chaining post processing to the ESTA algorithm has been named ESTA^C. Obviously greater CPU-time is required to use the chaining method, being that it is a post-processing phase. Furthermore using the chaining method two important features of the ESTA approach, the number of solved problems and the makespan, are preserved.

6 Experimental Evaluation

In this section we present the results obtained using either the resource envelope or the earliest start time approach embedded in the common framework introduced in Sect. 3.

For the evaluation we consider the Resource-Constrained Project Scheduling Problem with Minimum and Maximum time lags (RPCSP/max), which involves synchronizing the use of a set of renewable resources $R = \{r_1 \dots r_m\}$ to perform a set of activities $V = \{a_1 \dots a_n\}$ over time. The execution of each activity is subject to the following constraints:

- each activity a_j has a duration dur_{a_j}, a start time s_{a_j} and an end time e_{a_j} such that $e_{a_j} = s_{a_j} + dur_{a_j}$;
- each activity a_i requires the use of ru_{ik} units of the resource r_k.
- a set of temporal constraints c_k defined for an activities pair (a_i, a_j) of the form of $c_k^{min} \leq s_{a_j} - s_{a_i} \leq c_k^{max}$;
- each resource r_k has an integer capacity $cap_k \geq 1$;

A solution to a RCPSP/max is any consistent assignment to the start-time of all the activities in V which does not violate resource capacity constraints.

The results we will show have been obtained using the benchmarks defined in [9]. These consist of three sets of 270 instances of different size 10×5, 20×5 and 30×5 where the numbers represent respectively the number of activities and of resources involved. All algorithms presented in this paper are implemented in C++ and the CPU times presented in the following tables are obtained on a Pentium III-500 Mhz processor under Windows NT 4.0.

An initial comparison is presented according to the following parameters: (1) percentage of problems solved from a fixed set, (2) average CPU-time spent to solve instances of the problem, (3) average makespan and (4) the number

size	sol. (%)	makespan	CPU-time(sec.)	constraints
10	67.4	55.64	4.211	11.15
20	50.7	92.22	120.4	40.26
30	52.6	130.15	1376.4	87.53

Table 1. EBA

size	sol. (%)	makespan	CPU-time(sec.)	constraints
10	97.04	49.1	1.728	7.16
20	95.56	83.5	9.898	21.30
30	95.93	106.1	33.736	38.18

Table 2. ESTA^C

of leveling constraints posted to solve the problem. The last gives one estimate of the kind of flexible solution created (The higher the less desirable). We also consider the makespan because it gives the quality of the solution in the best case (no disruptions) possible. Later, in Section 6.1, we analyze the flexibility of the solutions achieved using each different approach. The parameters introduced in Sect. 2.1 will be used as a basis for that evaluation.

In Table 1 the results of the resource envelope-based approach without MCS filtering, EBA, are shown. Comparing these values with those obtained with ESTA^C, Table 2, it can be seen that the EBA approach is actually quite ineffective. It solves significantly fewer problems than ESTA^C in each problem set and incurs higher CPU times.

A significant drawback of using the resource envelope is its high associated computational cost. Indeed, the computation of the envelope implies that it is necessary to solve a Max-Flow problem for each time-point. As indicated in [6], this leads to an overall complexity of $O(n^4)$ which can be reduced to $O(n^{2.5})$ in practical cases. These computational requirements present a formidable barrier to effective application of the resource envelope. In point of fact, use the resource envelope within a scheduling problem solver requires recomputation of the envelope at each step of the search.

Comparison with the results obtained with the ESTA^C algorithm highlight a further negative aspect of the EBA approach: EBA consistently adds a larger set of leveling constraints than ESTA^C in generating a solution. In fact, this result could have been predicted. The ESTA^C approach, indeed, posts a set of "implicit" constraints each time the profile is computed: each activity has to start at its earliest start-time. These constraints temporarily restrict the solution's temporal flexibility (for the purpose of computing resource profiles). This avoids having to take into account all the possible temporal configurations of the set of the activities, and it follows that a smaller set of constraints are necessary to order them.

By adding MCS filtering to the EBA search configuration, we obtain a noticeable improvement of the result. Tables 3 and 4 represent respectively the results obtained using the linear and the quadratic sampling versions of MCS filtering. The use of MCS linear sampling gives an overall improvement of the ba-

size	sol. (%)	makespan	CPU-time(sec.)	constraints
10	76.7	54.83	10.29	10.86
20	69.3	99.88	162.95	30.88
30	66.7	132.7	1432.1	62.84

Table 3. EBA + MCS linear sampling

size	sol. (%)	makespan	CPU-time(sec.)	constraints
10	96.7	57.71	10.78	12.36
20	86.3	106.05	205.76	34.74
30	81.1	143.76	2101.2	65.74

Table 4. EBA + MCS quadratic sampling

sic approach. Although the results are still worse than the ESTA^C case, the MCS linear gives better values than the simple EBA with respect to all performance parameters. The use of the quadratic MCS version gives considerable further improvement (Table 4) with respect to the number of problems solved, and indeed the performance along this dimension is closer to that achieved with the ESTA^C approach. This could be predicted from the fact that the quadratic version takes a larger sample than the linear version from the space of the MCS. On the other hand, EBA with quadratic MCS has a negative impact both the CPU-time and the number of leveling constraints added. The first aspect follows from use of the more expensive MCS quadratic sampling procedure in conjunction with the resource envelope computation. The second aspect, instead, as suggested above, is a consequence of the nature of approaches that attempt to retain maximal temporal flexibility. Finally, Tables 5 and 6 present the results obtained using the two MCS sampling methods in conjunction with the earliest start time approach. Here we see only slight improvement over the basic ESTA^C procedure.

6.1 Flexibility

This section analyzes the flexibility of the solutions obtained using the resource envelope or the earliest start time profile to guide the algorithm. For each benchmark problem set we present the average value. Moreover taking into account that different sets of problems were solved by each approach we only consider the subset of problem instances solved by all six approaches. Table 7 presents the results of the six different approaches according to the parameters described in Sect. 2.1.

First consider the metric (1), which reflects the degree of "fluidity" of the generated solutions. The basic EBA approach tends to produce more robust solutions along this dimension when it is able to generate a solution. On the other hand since EBA is managing the total set of possible solutions, search heuristics for conflict selection and resolution generally provide less leverage (see Table 1) than in ESTA^C and there is greater chance of not finding a solution. Furthermore, even as the introduction of MCS filtering increases the percentage of problems solved, it also leads to solutions that on average are less fluid. It thus appears

size	sol. (%)	makespan	CPU-time(sec.)	constraints
10	98.15	48.55	1.832	3.03
20	96.67	82.97	13.37	11.01
30	97.04	106.35	86.220	22.33

Table 5. ESTA^C+MCS linear sampling

size	sol. (%)	makespan	CPU-time(sec.)	constraints
10	98.15	48.58	1.846	3.03
20	96.30	83.08	14.268	10.91
30	97.41	106.01	125.52	22.38

Table 6. ESTA^C+MCS quadratic sampling

that the heuristic bias introduced by MCS filtering has both positive and negative aspects, illustrating the difficult challenge associated with injecting heuristic guidance into the EBA search procedure. This behavior is not observed in the case of ESTA^C: indeed all three algorithms produce solutions with essentially the same fluidity.

To quantify the impact of possible disruption during the execution of the schedule, a second metric (2) was introduced in Sect 2.1. In the current evaluation we consider that the probability that a disruption occurs during the execution of an activity a_i is related to its duration and to the overall duration of the solution (mk), $Pr_{disr}(a_i) = \frac{dur_{a_i}}{mk}$. Furthermore for computing the number of changes we assume the biggest shift Δ_i possible for activity a_i in the worst case, that is, $num_{changes}(a_i, let(a_i) - eet(a_i))$. Then we can re-write the (2) as:

$$dsrp = \frac{1}{N} \sum_{i=1}^{N} \frac{dur_{a_i}}{mk} \times \frac{let(a_i) - eet(a_i)}{num_{changes}(a_i, let(a_i) - eet(a_i))} \tag{4}$$

Examining the values in the table, we see that along this dimension the ESTA^c approaches dominate the EBA approaches across all problem sets.

Final remarks. Considering the philosophies behind the different approaches that have been evaluated, one would expect that those that manage the knowledge of all the possible temporal allocations would provide the most effective basis for generating flexible solutions. As a matter of fact, we have shown a two step procedure for computing a fixed time schedule and translating it into a flexible solution to be a more effective approach. The first step allows advantage to be taken of the effectiveness of a fixed time scheduling approach (i.e., makespan and CPU time minimization), while the second step, has been shown to be capable of re-instating temporal flexibility in a way that preserves the qualities of the fixed time solution.

	fluidity			disruptibility		
	10	20	30	10	20	30
EBA	28.69	31.32	33.35	8.90	12.74	18.21
EBA+MCS linear	27.05	25.78	25.76	8.74	12.18	17.27
EBA+MCS quadratic	25.84	24.14	22.35	8.23	12.73	18.57
ESTA^C	29.18	29.49	28.09	10.20	16.38	23.80
ESTA^C+MCS linear	29.20	29.97	28.45	10.21	16.49	24.90
ESTA^C+MCS quadratic	29.20	30.05	28.06	10.20	15.34	24.31

Table 7. Fluidity & Disruptibility.

7 Conclusion and Future Work

In this paper, we have investigated two approaches for generating schedules that retain temporal flexibility and possess good robustness properties. Such flexible solutions promote an ability to react to exogenous events with minimal solution change and without external assistance. To support assessment of schedules from this perspective, we first defined measures of solution robustness relating to fluidity and disruptibility. We then developed two alternative approaches to constructing a flexible schedule: one based on use of the resource envelope introduced in [6] (named EBA) and the other based on use of the earliest start time profile [1]. The latter approach also involved the definition of a post processing method to transform a fixed-times schedule into a flexible schedule (the complete approach has been named ESTA^C). To provide a basis for comparative analysis, both approaches were formulated within a common framework.

Analyzing initial performance results obtained with both procedures, we found that EBA was able to solve significantly smaller numbers of problems than ESTA^C at much higher computational cost per solution. To improve EBA's performance, we incorporated two approximate methods for generating Minimal Conflict Sets (MCS): linear and quadratic sampling. The use of these methods did increase the number of solutions found but also increased the CPU-time and the number of leveling constraints posted. And, in all cases, ESTA^C continued to outperform EBA across all performance criteria. In analyzing the robustness of generated solutions, we found that the basic EBA procedure in fact produced solutions with greater fluidity when it was able to find a solution. However, as MCS sampling was incorporated and the number of problems solved increased, the fluidity of generated solutions simultaneously degraded, below the measured fluidity of schedules produced by ESTA^C. With regard to disruptibility, ESTA^C schedules dominated in all cases. Overall, ESTA^C was found to be a much more effective procedure.

Different aspects of the resource envelope approach and the earliest start time approach warrant further investigation. The former would benefit considerably from a more efficient envelope computation, considering that it is called into play extremely often. In the case of ESTA^C the algorithm for translating a fixed time solution into a flexible solution might be improved through use of more extended, local search techniques.

Acknowledgements

Stephen F. Smith's work is supported in part by the Department of Defense Advanced Research Projects Agency (DARPA) and the US Air Force Research Laboratory under contracts F30602-00-2-0503 and F30602-02-2-0149, and by the CMU Robotics Institute. Amedeo Cesta, Angelo Oddi and Nicola Policella's work is partially supported by ASI (Italian Space Agency) under project ARISCOM (Contract I/R/215/02). Nicola Policella is currently supported by a Scholarship from CNR on Information Technology. This work has been developed during Policella's visit at the CMU Robotics Institute as a visiting student scholar. He would like to thank the members of the Intelligent Coordination and Logistics Laboratory for support and hospitality.

References

1. A. Cesta, A. Oddi, and S. F. Smith. Profile Based Algorithms to Solve Multiple Capacitated Metric Scheduling Problems. In *Proceedings of the* 4^{th} *International Conference on Artificial Intelligence Planning Systems AIPS-98*, 1998.
2. A. Cesta, A. Oddi, and S. F. Smith. A Constraint-based method for Project Scheduling with Time Windows. *Journal of Heuristic*, 2002.
3. A.J. Davenport, C. Gefflot, and J.C. Beck. Slack-based Techniques for Robust Schedules. In *Proceedings of* 6^{th} *European Conference on Planning, ECP-01*, 2001.
4. M. Drummond, J. Bresina, and K. Swanson. Just-in-Case Scheduling. In *Proceedings of the* 12^{th} *National Conference on Artificial Intelligence, AAAI-94*, pages 1098–1104. AAAI Press, 1994.
5. M.L. Ginsberg, A.J. Parkes, and A. Roy. Supermodels and Robustness. In *Proceedings of the* 15^{th} *National Conference on Artificial Intelligence, AAAI-98*, pages 334–339. AAAI Press, 1998.
6. N. Muscettola. Computing the Envelope for Stepwise-Constant Resource Allocations. In *Principles and Practice of Constraint Programming,* 8^{th} *International Conference, CP 2002*, volume 2470 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2002.
7. P.Laborie and M. Ghallab. Planning with Sharable Resource Constraints. In *Proceedings of the* 14^{th} *International Joint Conference on Artificial Intelligence, IJCAI-95*, pages 1643–1649, 1995.
8. H. H. El Sakkout and M. G. Wallace. Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling. *Constraints*, 5(4), 2000. Special Issue on Industrial Constraint-Directed Scheduling.
9. C. Schwindt. A Branch and Bound Algorithm for the Resource-Constrained Project Duration Problem Subject to Temporal Constraints. Technical Report 544, Institut fur Wirtschaftstheorie und Operations Research, Universit at Karlsruhe, 1998.
10. S. F. Smith. OPIS: A Methodology and Architecture for Reactive Scheduling. In M. Fox and M. Zweben, editors, *Intelligent Scheduling*. Morgan Kaufmann, 1994.
11. S. F. Smith and C. Cheng. Slack-based Heuristics for Constraint Satisfactions Scheduling. In *Proceedings of the* 11^{th} *National Conference on Artificial Intelligence, AAAI-93*, pages 139–144. AAAI Press, 1993.

Preferences and Uncertainty in Simple Temporal Problems

Francesca Rossi[1], Brent Venable[1], and Neil Yorke-Smith[2]

[1] University of Padova, Italy {frossi,kvenable}@math.unipd.it
[2] IC–Parc, Imperial College London, U.K. nys@icparc.ic.ac.uk

Abstract Simple Temporal Problems (STPs) are a tractable restriction of the framework of Temporal Constraint Satisfaction Problems. Their expressiveness has been extended in two ways. First, to account for uncontrollable events — called Simple Temporal Problems with Uncertainty (STPUs) — and second, to account for soft preferences — called Simple Temporal Problems with Preferences (STPPs). The motivation for both extensions is from real-life problems; and indeed such problems may well necessitate *both* preferences and uncertainty. To meet this need we define Simple Temporal Problems with Preferences and Uncertainty (STPPUs). We extend the notions of controllability to STPPUs, and describe methods to determine whether these properties hold.

1 Motivation and Background

Research on temporal reasoning, once exposed to the difficulties of real-life problems, can be found lacking both expressiveness and flexibility. Planning and scheduling for satellite communication [7], for example, involves not only quantitative temporal constraints between events and qualitative temporal ordering of events, but also soft temporal preferences and contingent events over which the agent has no control. For example, communication should be avoided during manoeuvers, if possible; while communication durations, which depend on climatic conditions, are not under the agent's direct control.

To address the lack of expressiveness of hard constraints, preferences can be added to the framework; to address the lack of flexibility to contingency, handling of uncertainty can be added. Some real-world problems, however, have need for both. It is this requirement that motivates us here.

1.1 Temporal Constraint Satisfaction Problems with Preferences

In a temporal constraint problem, variables denote timepoints or intervals, and constraints represent the possible temporal relations between them. Dechter et. al. [2] introduced quantitative *Temporal CSPs* (TCSPs) and the restricted subclass of *Simple Temporal Problems* (STPs). In a STP, variables x_i represent timepoints (events) and constraints represent the relations between them. Because the constraints are restricted to a single interval — they have the form

$l_{ij} \leq x_j - x_i \leq u_{ij}$ — a STP can be solved in polynomial time. By solved, we mean that consistency is decided and the minimal network obtained; applying path consistency suffices for this. In contrast, general TCSPs are NP-complete.

To address the lack of expressiveness in standard STPs, Khatib et. al. [4] introduced *Simple Temporal Problems with Preferences* (STPPs), which merge temporal CSPs with semiring-based soft constraints [1]. Alongside the hard temporal constraints of a STP, soft temporal constraints are specified by means of a *preference function* on an interval, $f : I \rightarrow A$, where $I = [l_{ij}, u_{ij}]$ and A is a set of preference values, part of a semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$.[3]

In general, solving a STPP is NP-complete. However, by making the following assumptions, solving is polynomial in the number of timepoints: (1) the preference functions are semi-convex[4], (2) the semiring multiplicative operator is idempotent, and (3) the preferences are totally ordered.

Rossi et. al. [9, 8] present two solvers for STPPs. Both find the globally optimal solution in terms of maximising the minimal local preference values. The first, Path-solver, enforces path consistency in the constraint network, then takes the sub-interval on each constraint corresponding to the best preference level. This gives a standard STP, which is then solved for the first solution by backtrack-free search. The complexity is polynomial, but the performance can be poor because a pointwise (discrete) representation is used for the intervals and the preference functions. The second solver, Chop-solver, is less general but more efficient. It finds the maximum level α at which the preferences can be 'chopped', i.e. the intervals are reduced to the set $\{x : x \in I, f(x) \geq \alpha\}$ of values mapped to at least α by the preference functions. This set is a simple interval for each I, provided the above assumptions hold. Hence we obtain a standard STP, say STP_α. By binary search, the solver finds the maximal α for which STP_α is consistent. The solutions of this STP are the solutions of the original STPP. A third solver, that obtains Pareto optimal solutions, is in [3].

1.2 Simple Temporal Problems with Uncertainty

To address the lack of flexibility in execution of standard STPs, Vidal and Fargier [10] introduced *Simple Temporal Problems under Uncertainty* (STPUs).

Here, as in a STP, the activities have durations specified by intervals. The durations fall into two classes, *requirement* and *contingent*. The former, as in a STP, can be decided, but the latter are decided by 'Nature' — we have no control over when the task will end; we just observe it rather than execute it. The only information known prior to observation is that Nature will respect the interval on the duration. Durations of contingent links are assumed independent.

[3] A semiring is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that: A is a set and $\mathbf{0}, \mathbf{1} \in A$; $+$ is commutative, associative and $\mathbf{0}$ is its unit element; $\times$ is associative, distributes over $+$, $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element. A *c-semiring* is a semiring in which $+$ is idempotent, $\mathbf{1}$ is its absorbing element and $\times$ is commutative.

[4] A semi-convex function is function with at most one peak: $f : I \rightarrow A$ is semi-convex if $\forall \alpha \in A$ the set of elements $\{x \in I | f(x) \geq \alpha\}$ forms a unique interval.

Controllability of a STPU is the analogue of consistency of a STP. Controllable means we have a way to execute the timepoints under our control, subject to all constraints. Three notions are proposed:

- A STPU is *strongly controllable* if there is a fixed execution strategy that works in all realisations. (A *realisation* is a possible outcome of the world, i.e. in this case, an observation of all contingent timepoints.)
- A STPU is *dynamically controllable* if there is a online execution strategy that depends only on observed timepoints in the past and that can always be extended to a complete schedule whatever may happen in the future.
- A STPU is *weakly controllable* if there is a viable global execution strategy: there exists at least one schedule for every realisation.

The three are ordered by their strength: Strong $\Longrightarrow$ Dynamic $\Longrightarrow$ Weak. The first requires no knowledge of the realisation, and checking it is in P. Checking the second, surprisingly, is also in P [6]. It is seen as the most realistic knowledge assumption in many practical cases, since it interleaves scheduling, observation and execution. The third requires a prior knowledge of the realisation, and checking it is co-NP complete [10].

In this paper we formally define a class of temporal constraint satisfaction problems that feature both preferences and uncertainty. For this class of problems we consider the equivalent of Strong, Weak and Dynamic Controllability. In particular we extend the notions of controllability and we give algorithms to check them. We show that adding preferences does not impact on the complexity of checking these types of controllability. We tackle the most difficult case, Dynamic Controllability with optimal preference levels, by giving a sound but incomplete algorithm.

2 STPs with Preferences and Uncertainty

Consider a temporal problem which features preferences, hard constraints and also uncertainty. Neither a STPP nor a STPU is adequate. Therefore we propose what we will call a *Simple Temporal Problems with Preferences and Uncertainty.*

An informal definition of a STPPU is a STPP where timepoints are partitioned into two classes, requirement and contingent, as in a STPU. Since some timepoints are not controllable, the notion of consistency of a STP(P) is replaced by controllability. Every solution to the STPPU has a global preference value, as in a STPP, and we seek a solution which maximises this value.

More precisely, we can extend some definitions given for STPPs and STPUs to fit STPPUs in the following way. Following [10], we say *executable timepoints* are those points, b_i, whose date is assigned by the agent, while *contingent timepoints* are those points, e_i, whose uncontrollable date is assigned by the external world; a *generic timepoint* t_i is either an executable or a contingent timepoint. We say a *decision* $\delta(b_i)$ is a value assigned to an executable timepoint, while an *observation* $\omega(e_i)$ is a value assigned (by Nature) to a contingent timepoint; an *assignment* $\gamma(t_i)$ is a value assigned by either a decision to an executable timepoint or by an observation to a contingent timepoint.

Definition 1. – *A* soft requirement constraint r_{ij}, *on generic timepoints* t_i *and* t_j, *is a pair* $\langle I_{ij}, f_{ij}\rangle$, *where* $I_{ij} = [l_{ij}, u_{ij}]$ *such that* $l_{ij} \leq \gamma(t_j) - \gamma(t_i) \leq u_{ij}$, *and* $f_{ij} : I_{ij} \to A$ *is a* requirement preference function *that maps each element of the interval into an element of the preference set of a semiring;*
– *A* soft contingent constraint g_{ij} *is a pair* $\langle \hat{I}_{ij}, \hat{f}_{ij}\rangle$ *where* $\hat{I}_{ij} = [\hat{l}_{ij}, \hat{u}_{ij}]$ *such that* $\hat{l}_{ij} \leq \gamma(t_j) - \gamma(t_i) \leq \hat{u}_{ij}$, *and* $\hat{f}_{ij} : \hat{I}_{ij} \to A$ *is a* contingent preference function.

Although the definitions of soft requirement and contingent constraints are similar, there is an important semantic difference between them. Preference on a requirement constraint is 'effective': the agent can act to maximise this preference. In contrast, preference on a contingent constraint is 'reflective': it shows the agent's opinion of a value Nature chooses. For instance, while we cannot control climatic conditions, we do prefer a short communication duration; a soft contingent constraint reflects this preference.

We can now state formally the definition of a STPPU, combining preferences from the definition of STPPs with contingency from the definition of STPUs:

Definition 2 (STPPU). *A* Simple Temporal Problem with Preferences and Uncertainty (STPPU) *is a tuple* $P = (N_e, N_c, L_r, L_c, S)$ *where:*

– N_e *is the set of executable timepoints;*
– N_c *is the set of contingent timepoints;*
– L_r *is the set of soft requirement constraints over* S*;*
– L_c *is the set of soft contingent constraints over* S*;*
– $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ *is a c-semiring*

An example of a simple STPPU is depicted overleaf in Figure 1.

In order to analyse the solutions to a STPPU, we need further preliminary definitions. We say a *requirement duration* γ_{ij} is any point in interval I_{ij} of requirement constraint r_{ij}, i.e. $\gamma_{ij} = \gamma(t_j) - \gamma(t_i)$; while a *contingent duration* ω_{ij} is any point in interval $\hat{I}_{ij}$ of contingent constraint g_{ij}, i.e. $\omega_{ij} = \gamma(t_j) - \gamma(t_i)$. We say a *control sequence* δ of the STPPU is an assignment of the executable time points $\delta = \{\delta(b_1), \ldots, \delta(b_n)\}$. If it is an assignment to all the executable timepoints, the control sequence is said to be *complete*, otherwise *partial*. Every control sequence is associated with a preference value, $\text{pref}(\delta) = \prod_{r_{ij}|\exists\delta(b_i),\delta(b_j)} f_{ij}(\delta(b_j) - \delta(b_i))$, where $\prod$ represents the multiplicative operator of the semiring.

Finally, the *space of complete realisations* Ω of STPPU is the Cartesian product of all contingent intervals, i.e. $\Omega = [\hat{l}_1, \hat{u}_1] \times \cdots \times [\hat{l}_G, \hat{u}_G]$. For all $\omega \in \Omega$, the *projection* P_ω of STPPU P is the STPP obtained replacing in each soft contingent constraint g_k, the interval $\hat{I}_k$ with $[\omega_k, \omega_k]$. Every realisation is associated with a preference value, $\text{pref}(\omega) = \prod_{g_{ij}|\exists\omega(e_i),\omega(e_j)} \hat{f}_{ij}(\omega(e_j) - \omega(e_i))$.

Definition 3 (Schedule). *A* schedule T *is a complete assignment to all the timepoints of STPPU* P*; a schedule identifies an assignment* γ_T *or, more precisely, a control sequence* δ_T *and a realisation* $\omega_T = \{\omega_{ij}^T\}$ *(we will write* $T =$

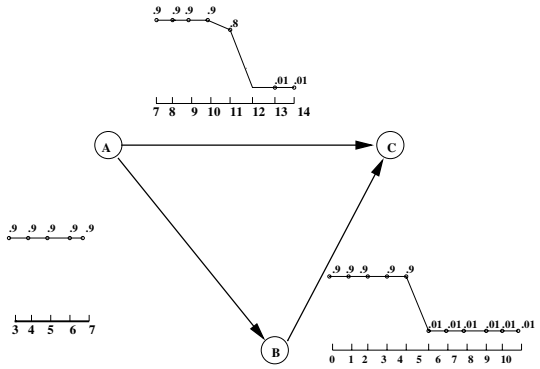


Figure 1. Example of a triangular STPPU

(δ_T, ω_T)). *Hence a schedule identifies a unique set of requirement durations* $\{\gamma_{ij}^T\}$ *and it is said to be* consistent *if* $\forall r_{ij}, f_{ij}(\delta_{ij}^T) > 0$ *and* $\forall g_{ij}, \hat{f}_{ij}(\omega_{ij}^T) > 0$. *Every schedule is associated with a preference:* $\text{pref}(T) = \text{pref}(\delta_T) \times \text{pref}(\omega_T)$.

We can now give the first two types of controllability which take into account both contingency and preferences. The idea is first to ensure contingency is met, and only second to maximise the preference. Specifically, for each notion of controllability, the preferences are combined with the contingency in two ways. In *optimal* controllability we require a schedule with maximal preference; in α controllability we require a schedule with preference at least α.

Definition 4 (Optimal Strong Controllability). *A STPPU is* Optimally Strongly Controllable *iff there exists a control sequence* δ *such that for all* $\omega \in \Omega$, $T = (\delta, \omega)$ *is a consistent schedule for* P_ω, *and* $\text{pref}(T)$ *is optimal (i.e. there is no other schedule* T' *consistent with projection* P_ω *such that* $\text{pref}(T') > \text{pref}(T)$*).*

Definition 5 (α-Strong Controllability). *A STPPU is* α-Strongly Controllable*, with* $\alpha \in A$*, iff there exists a control sequence* δ *such that for all* $\omega \in \Omega$, $T = (\delta, \omega)$ *is a consistent schedule for projection* P_ω, *and* $\text{pref}(T) \geq \alpha$.

Definition 6 (Optimal Weak Controllability). *A STPPU is* Optimally Weakly Controllable *iff for all* $\omega \in \Omega$ *there exists a control sequence* δ_ω *such that* $T = (\delta_\omega, \omega)$ *is a consistent schedule for projection* P_ω, *and* $\text{pref}(T)$ *is optimal for* P_ω.

Definition 7 (α-Weak Controllability). *A STPPU is* α-Weakly Controllable*, with* $\alpha \in A$*, iff for all* $\omega \in \Omega$ *there exists a control sequence* δ_ω *such that* $T = (\delta_\omega, \omega)$ *is a consistent schedule for projection* P_ω, *and* $\text{pref}(T) \geq \alpha$.

Intuitively, under Optimal Controllability we may obtain substantially different preference values on different constraints; while under α-Controllability we obtain more uniform values, but potentially far from the optimum for any constraint. Which notion will be more useful will depend on the problem domain.

Before we are ready to define our extensions to Dynamic Controllability, we need a few further preliminaries. We say the *executed control sequence at t* of a complete control sequence δ is the partial sequence $\delta_{\prec t,\delta} \subseteq \delta$ executed so far at time t, defined by: $\delta_{\prec t,\delta} = \{\delta(b_i) | \delta(b_i) \leq t\}$. Similarly, the *observed realisation at t* associated with $\delta_{\prec t,\delta}$ and ω is the partial realisation $\omega_{\prec t,\delta,\omega} \subseteq \omega$ such that: $\omega_{\prec t,\delta,\omega} = \{w_{ij} \in \omega | \delta_{\prec t,\delta}(b_i) + w_{ij} \leq t\}$. Finally, the *subspace* of complete realisations *compatible* with a partial realisation ω_p is $\Omega_{\omega_p} = \{\omega' \in \Omega | \omega_p \subseteq w'\}$.

Definition 8 (Optimal Dynamic Controllability). *A STPPU is* Optimally Dynamically Controllable *iff for all $\omega \in \Omega$ there exists a control sequence δ such that $\forall t \in \delta, \forall \omega' \in \Omega_{\omega_{\prec t,\delta,\omega}}$, $\exists \delta'$, such that (1) $\delta_{\prec t,\delta} \subseteq \delta'$, and (2) schedule $T = (\delta', \omega')$ is consistent with $P_{\omega'}$, and (3)* $\text{pref}(T)$ *is optimal.*

Definition 9 (α-Dynamic Controllability). *A STPPU is* α-Dynamically Controllable *iff for all $\omega \in \Omega$ there exists a control sequence δ such that $\forall t \in \delta, \forall \omega' \in \Omega_{\omega_{\prec t,\delta,\omega}}$, $\exists \delta'$, such that (1) $\delta_{\prec t,\delta} \subseteq \delta'$, and (2) schedule $T = (\delta', \omega')$ is consistent with $P_{\omega'}$, and (3)* $\text{pref}(T) \geq \alpha$.

In the following sections, we briefly consider Strong and Weak Controllability. Then, since it is the most interesting case, we consider Dynamic Controllability in greater detail. In Section 6 we show that ODC is the most involved case.

3 Strong Controllability

In this section we describe algorithms that check, in polynomial time, whether a STPPU P is Optimally Strongly Controllable (OSC), and whether P is α-Strongly Controllable (α-SC).

The algorithms we propose rely on two known algorithms. The first of these is Path-solver [8], which enforces path consistency on a STPP. The second algorithm is Strong-Controllability [11], which checks if a STPU is Strongly Controllable. In order to use these algorithms we require the preference functions to be semi-convex. For convenience, we will also assume that the semiring underlying our constraint problems is the *fuzzy semiring* $S_{FCSP} = \{[0,1], max, min, 0, 1\}$; this assumption is not restrictive [4].

The main idea is to consider P as a STPP, by ignoring the uncertainty, and as a STPU, by neglecting the preference. More precisely, any STPPU can be treated as a STPP by ignoring the fact that some constraints are contingent. Let IU ('Ignore Uncertainty') be a function that maps a STPPU $P = (N_e, N_c, L_r, L_c, S)$ into STPP $IU(P) = \langle I, f \rangle$, where the set of intervals I is the set of all the intervals of soft constraints in L_r and L_c, and preference function $f : I \to A$ acts on each interval as the preference function of the soft constraint in P [8].

3.1 Optimal Strong Controllability

For checking SC, since we are not interested in obtaining an actual solution, we only need apply the first part of Path-solver. We will call this sub-algorithm

Soft-PC-2; it takes a STPP and enforces path consistency. As a result, it squeezes some intervals and lowers some preference functions. All the preference functions reach the same maximum preference level, which we will call *opt*.

Soft-PC-2 returns a STPP that has interesting features. First, the intervals consist of a minimal STP (i.e. a problem containing only points that appear in at least one solution). Second, the sub-STP consisting of the sub-intervals mapped by the preference functions into *opt* is minimal as well, and all its solutions are optimal solutions of the original STPP. We will use these properties when considering dynamic controllability.

We name P_{opt} the STPU obtained by considering the sub-intervals mapped into *opt* on all the requirement constraints after Soft-PC-2, and the original intervals on all the contingent constraints. The semi-convexity of the preference functions guarantees that P_{opt} is a STPU and not a TCSPU. We will call OPT the procedure that, given as input a path consistent STPP, returns a STPU with the structure we have just described.

If any contingent constraint is squeezed when enforcing path consistency, we can conclude that the problem is not pseudo-controllable [5], and hence not SC. Further, the following theorem allows us to conclude that it cannot be OSC. All proofs have been omitted for lack of space.

Theorem 1. *If a STPPU P is OSC, then the STPU obtained by simply neglecting preference functions on all the constraints is SC. However, the converse in general is false.*

To summarise, the algorithm we propose for checking Optimal Strong Controllability of a STPPU P first applies Soft-PC-2 to $IU(P)$. If any contingent interval is squeezed during the process then the algorithm stops since the problem cannot be OSC. Otherwise it extracts P_{opt} from path consistent $IU(P)$, and runs Strong-Controllability on P_{opt}. We call the method Path-OSC. The following result guarantees that the algorithm is both sound and complete:

Theorem 2. *STPPU P, with semi-convex preference functions, is OSC iff the corresponding STPU P_{opt} is SC.*

Path-OSC has polynomial time complexity. The complexity of Soft-PC-2 is the same as Path-solver: $O(n^3 \times R \times l)$, where $n = |N_r| + |N_c|$, R is the maximum range of an interval, and l is the number of preference levels. Procedures IU and OPT are linear in the total number of constraints, which is $O(n^2)$. Strong-Controllability has the same complexity as PC-2: i.e. $O(n^3 \times R)$. Hence Path-OSC has total complexity of $O(n^3 \times R \times l)$. Note that this is in line with results on STPUs [11]. In fact, just like SC for STPUs, the complexity of checking OSC of a STPPU has the same complexity as enforcing path consistency.

It is worth noting that another possibility is to combine Strong-Controllability with Chop-solver [8] rather than Path-solver. This gives a similar algorithm to Path-OSC. It has the same complexity, but in practice may exhibit better performance, since Chop-solver has better practical performance than Path-solver.

3.2 α-Strong Controllability

We now tackle the problem of verifying whether a STPPU P is α-SC or not. It may seem that OSC is equivalent to *opt*-SC (i.e. α-SC with $\alpha = opt$, where *opt* is the maximum preference level at which Chop-solver finds a consistent STP). However this is not the case. OSC means that there exists a control sequence that, when completed with a realisation, is optimal for the projection corresponding to that realisation. α-SC, however, imposes that the completed control sequence must have a preference at least α on all the projections. Nonetheless, the analogue of Theorem 1 does hold:

Theorem 3. *If a STPPU P is α-SC, then the STPU obtained by neglecting preference functions on all the constraints is SC. However, the converse in general is false.*

We observe that no STPPU can ever be α-consistent for any $\alpha > \alpha^* = \min_{ij|\exists g_{ij}} \hat{f}_{ij}(\omega_{ij})$. To see this, suppose ω is a realisation in which some constraint has preference smaller than α. Then a projection corresponding to ω has only solutions with preference strictly less than α.

With this observation, it is possible to put α-SC of an STPPU P in one-to-one correspondence to the SC of a related STPU P^α. P^α is the problem obtained chopping the preference functions of P at level α (ignoring the uncertainty). Since $\alpha \leq \alpha^*$, contingent constraints maintain their intervals after the chop.

Theorem 4. *STPPU P is α-SC iff the corresponding STPU P^α is SC.*

To summarise, to check α-SC we propose two steps. Chop $IU(P)$ at level α; then restore information about contingent links, giving P^α, and on P^α run Strong-Controllability. We call the method Chop-α-SC; its complexity is $O(n^3 \times R)$.

A final query to answer is: what is the highest level α at which P is α-SC? (Note that in general $\alpha \ll opt$.) We propose a binary search algorithm very similar to Chop-solver: the only modification is to replace, at every chop level, PC-2 with Strong-Controllability. We call the method Max-α-SC; its complexity is $O(p \times n^3 \times R)$, where p is proportional to the search precision required.

4 Weak Controllability

We will now consider the impact of adding preferences to the issue of Weak Controllability. The following theorem states that Optimal Weak Controllability (OWC) and WC are related in the opposite way to OSC and SC:

Theorem 5. *A STPPU P is OWC if the STPU obtained by neglecting preference functions on all the constraints is WC. The converse in general is false.*

The converse fails in general because we must take into account the possibility of mapping some elements of the intervals into **0**. However if all the elements are mapped into strictly positive preferences, then the converse does hold.

Theorem 5 allows us to conclude that to check OWC, it is enough to apply algorithm Weak-Controllability proposed in [11].

To check α-WC we have two different approaches. The first approach is to chop the STPPU at level α and then to apply Weak-Controllability to the STPU obtained. The second possibility is to use the fact that a STPU is WC iff all the projections P_ω with $\omega \in \{\hat{l}_1, \hat{u}_1\} \times \cdots \times \{\hat{l}_h, \hat{u}_h\}$, where h is the number of contingent constraints, are consistent STPs [11]. Using this, the second approach is to chop each projection P_ω at level α and then to check the consistency of the derived STP. The complexity of both algorithms is exponential in the number of contingent constraints h: $O(2^h \times n^3 \times R)$.

5 α-Dynamic Controllability

Our approach to Dynamic Controllability will be the same as in the last two sections. We reduce the STPPU to chosen STPPs and STPUs, in order to leverage existing algorithms. We begin with α-DC because it shares more in common with the previous cases than does ODC.

As before, we first consider checking α-DC of a STPPU P, and then (if the property holds) consider finding the maximum α at which P is α-DC. We start with the analogue of Theorem 3:

Theorem 6. *If a STPPU P is α-DC, then the STPU obtained by neglecting preference functions on all the constraints is DC. However, the converse in general is false.*

Similar to SC, we can put α-DC of a STPPU P in one-to-one correspondence to DC of a related STPU P^α. Recall that P^α is the problem obtained chopping P at level α, then restoring the contingency information. Again, if $\alpha \leq \alpha^* = \min_{ij|\exists g_{ij}} \hat{f}_{ij}(\omega_{ij})$, contingent links will maintain their intervals after the chop.

Theorem 7. *STPPU P is α-DC iff the corresponding STPU P^α is DC.*

Hence we can define an algorithm Chop-α-DC, similar to Chop-α-SC. Checking DC of the STPU involved in this algorithm, however, is more complicated than checking SC was; we use the polynomial 3DC+ algorithm proposed in [6].

Secondly, to determine the highest level α at which P is α-DC, we proceed similarly to α-SC. The resulting algorithm Max-α-DC performs a binary search for the highest level α at which the problem is α-DC; as before, its complexity is polynomial in the size of the problem and the required precision.

6 Optimal Dynamic Controllability

In this and the next section we consider how to check if a STPPU is Optimally Dynamically Controllable. Our first step is to examine ODC of triangular STPPUs in which only one constraint is contingent, the simplest situation. We refer to the generic example of a triangular STPPU shown in Figure 2.

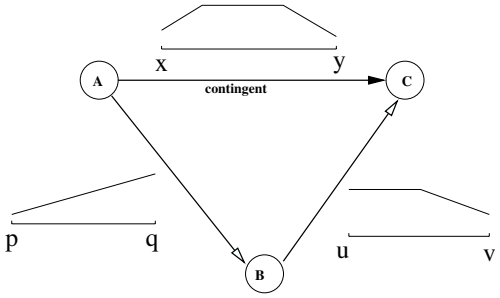


Figure 2. Triangular STPPU. Link AC is contingent.

We will consider networks to which Soft-PC-2 has been applied. As in the STPU case, we want to find (if any) the additional tightenings of the bounds of the intervals that must be obeyed by any optimal solution resulting from a dynamic strategy. But we also want to consider any changes in the preferences levels. Following [6], we will consider three cases involving the signs of the bounds $[u, v]$ of the BC constraint. For any variable X, we will write T_X to denote the time at which event X occurs.

6.1 Follow Case

If $v < 0$ then we are in the *Follow Case*: B will always follow C. This means when the agent must decide at what time B should occur, it will already know that C has occurred at time T_C, and it will know what preference is associated with $T_C - T_A$ on constraint AC. The Follow case is thus the simplest of the three; the following theorem shows that path consistency suffices to decide ODC.

Theorem 8. *Consider a triangular STPPU P, with interval $[u, v]$, $v < 0$, on the BC constraint. If P is path consistent then it is ODC.*

To ease execution, after checking controllability in general, we benefit if the checking algorithm returns a minimal network. Recall that minimality holds if only those elements belonging to at least one optimal solution are retained. In the Follow case, as in the two following, we will present procedures that, when applied to an ODC network, reduce it to its minimal form.

To obtain minimality in the Follow case, we perform a binary search for the highest level, α_{max}, at which chopping constraints AB and BC does not squeeze the contingent interval $[x, y]$. We denote this procedure by $BS(\alpha_{max}, P)$. Note that the binary search can start at level $\alpha_{min} = \min_{t \in [x,y]} \hat{f}_{AC}(t)$, i.e. the minimum preference assigned to any of the elements of the contingent interval.[5]

Theorem 9. *P' is minimal, unless some constraint has an empty interval.*

[5] This holds since no elements mapped in a preference strictly less than α_{min} correspond to an optimal solution.

To summarise, in the Follow case we can reduce $[p, q]$ and $[u, v]$ to $[p', q']$ and $[u', v']$ such that $\forall t_1 \in [p', q']$ and $t_2 \in [u', v']$, $f_{AB}(t_1) \geq \alpha_{max}$ and $f_{BC}(t_2) \geq \alpha_{max}$ and $\alpha_{max} \geq \alpha_{min}$, obtaining STPPU P'.

6.2 Precede Case

If $u \geq 0$ then we are in the *Precede Case*: B occurs before or simultaneously with C. In [6] it is shown that interval $[p, q]$ can be shrunk to $[y - v, x - u]$, and that if the STPU network is still pseudo-controllable then it is *safe*, and so DC.

Let us consider $\beta^{max} = \max_{t \in [y-v, x-u]} f_{AB}(t)$. All the elements of $[y-v, x-u]$ that are mapped into β^{max} satisfy the two following requirements: (1) each element is consistent with all the possible assignments to C, and (2) among the elements that have this property they are those with highest preference. We thus reduce $[p, q]$ to $[p', q']$ such that $\forall t \in [p', q']$, $f_{AB}(t) = \beta^{max}$.

After this reduction we must reapply Soft-PC-2 to see if any of the preferences on contingent link AC are lowered. If so, it means that the optimal solutions of the corresponding projections have been left out by the reduction. This in turn means that for such projections it is not possible to guarantee ODC.

The above reduction however is not enough to guarantee completeness of our algorithm for checking ODC: some elements belonging to $[p', q']$ might extend an observation on AC to an element on BC that has a lower preference. For example, consider the STPPU shown earlier in Figure 1, where $y - v = 4$ and $x - u = 7$ on the AB interval. Value $\beta_{max} = 0.9$ and interval $[p', q'] = [4, 7]$. However if the agent chooses a time for B that identifies point 6 on the AB interval, this will not be an optimal choice if the observation on AC is 11, since point 5 $(= 11 - 6)$ on the BC interval has a very low preference (0.01).

In general, since the agent must choose a time, T_B, for B before C occurs, when it chooses it must be sure, whatever time, T_C, *Nature* will assign to C, that T_B is consistent with T_C and is optimal. In our terms this means that $f_{AB}(T_B - T_A) \geq f_{AC}(T_C - T_A)$ and $f_{BC}(T_C - T_B) \geq f_{AC}(T_C - T_A)$.

Thus, to guarantee ODC, we might have to reduce further the interval $[p', q']$ on the AB constraint. Consider any preference α, assigned to some element of interval $[x, y]$ on the AC constraint. For each α we can consider the STP obtained by chopping the problem at level α (note that we chop the contingent link as well). After running path consistency we obtain intervals $[x_\alpha, y_\alpha]$ on AC and $[u_\alpha, v_\alpha]$ on BC. We then define $[p_\alpha, q_\alpha] = [y_\alpha - v_\alpha, x_\alpha - u_\alpha]$. Each interval $[p_\alpha, q_\alpha]$ contains the elements of AB that: (1) are consistent with *all* the elements of AC mapped into preferences at least α, and (2) in correspondence with any such elements on AC, identify an element on BC that has a preference at least α. Finally, we reduce AB to the intersection of the $[p_\alpha, q_\alpha]$ intervals, i.e. to the interval: $[p'', q''] = \bigcap_{\forall \alpha, \exists t \in [x,y], f_{AC}(t) = \alpha} [p_\alpha, q_\alpha]$.

We denote this procedure applied to our STPPU P with $M(P)$. The new interval $[p'', q'']$, found applying by M, will contain elements that are consistent with any element t of interval $[x, y]$. Moreover $\forall t'' \in [p'', q'']$, $\forall t \in [x, y]$, $f_{AB}(t'') \geq f_{AC}(t)$ and $f_{BC}(t - t'') \geq f_{AC}(t)$. Hence the agent may choose any

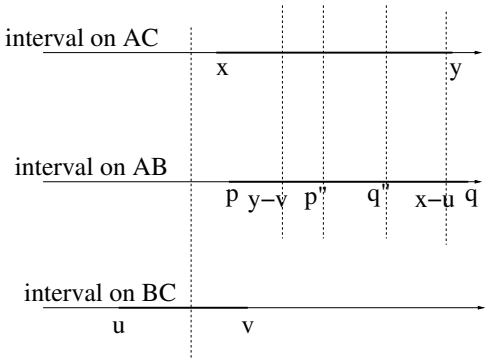


Figure 3. Intervals of the constraints in a triangular STPPU, Unordered case

points in $[p'', q'']$: whatever realisation will be observed in the future on AC, consistency and optimality are guaranteed. In the example shown in Figure 1, $[p'', q''] = [7, 7]$; thus the network is ODC. On the other hand, if interval $[p'', q'']$ is empty we can conclude that the network is not ODC. Notice how this reduction applied to an ODC triangular STPPU gives the minimal network.

To summarise, in the Precede case we first shrink $[p, q]$ to the sub-interval $[p', q']$, containing only elements mapped to β^{max} and belonging to sub-interval $[y - v, x - u]$. Then we reapply Soft-PC-2. If the propagation of the reductions cause any change on the AC constraint, we conclude that the triangular network is not ODC. Otherwise we reduce $[p', q']$ to $[p'', q'']$ by applying $M(P)$.

6.3 Unordered Case

If $u < 0$ and $v \geq 0$ then we are in the *Unordered Case*: B may or may not follow C. In [6] it is shown that B must wait until either C occurs or $y - v$ instants pass after A; this wait is denoted by $\langle C, y - v \rangle$. In a STPPU, where there are preferences, the agent must try to choose B maximising preferences on AB. The intervals of the three constraints are shown in Figure 3.

From the analysis of the hard constraint case in [6], we know that no element belonging to $[y - v, y]$ on the AC interval can be in the same solution as an element from $[p, y - v[$ on the AB constraint.[6] This means that for an optimal dynamic strategy to be feasible, running path consistency on STP Q_1, obtained considering interval $[y-v, y]$ on the AC constraint, $[y-v, q]$ on the AB constraint and $[u, v]$ on the BC constraint, and the restriction of the preference functions on these intervals, must not lower any preference on the AC constraint. If this first test fails then we can conclude that the triangular STPPU is not ODC.

The next step, if Soft-PC-2 does not change any preference on AC, is to look for those values on AB that are optimal and consistent for each projection. We will do this applying procedure M, as defined above, to Q_1. We, thus, find an

[6] With notation $[a, b[$ we are considering the continuous representation; in the discrete case we will write $[a, b - 1]$.

interval $[p'', q''] \subseteq [y - v, q]$ on AB. As before, if $[p'', q'']$ is empty we can conclude that the network is not ODC. If $[p'', q'']$ is non-empty we can lower the upper bound on AB from q to q'', and, consequently, raise if needed the lower bound on BC from x to $x - q''$.

At this point, we see that in a dynamic strategy, B will have to wait either for C to occur or for $p'' \geq y - v$ after A. This implies that if we consider the sub-interval $[x, p''[$ on AC and $[p, q'']$ on AB then the only possibility is for B to occur after C, just as in the Follow case. This means that running Soft-PC-2 on STP Q_2 with sub-interval $[x, p''[$ on AC and $[p, q'']$ on AB, and $[u, 0[$ on BC, and the preference functions restricted to these intervals, should not lower any preference on the AC. If running Soft-PC-2 does produce any change on AC, the network is not ODC; otherwise we can conclude the triangular network is ODC, provided the $\langle C, p'' \rangle$ wait on AB is satisfied.

In order to have also minimality of the network, a further step is necessary. Again like in the Follow case, we must search for the α_{max} level in Q_2, discarding all elements that do not appear in any optimal solution. Running procedure $BS(\alpha_{max}, Q_2)$ might shrink intervals $[p, q'']$ on AB to $[p_2, q_2]$. We raise the lower bound on AB to p_2, while leaving the upper bound at q''. We can consistently lower, if needed, the upper bound on AB to $y - p_2$.

```
ODC-UNORDERED(triangular STPPU P)
 1  STPP J ← IU(P)
 2  SOFT-PC-2(Q_1)
 3  if AC is unchanged
 4     then [p'', q''] ← M(Q_1)
 5          if [p'', q''] ≠ ∅
 6             then ub_AB ← q'', lb_BC ← x − q''
 7                  SOFT-PC-2(Q_2)
 8                  if AC is unchanged
 9                     then [p_2, q_2] ← BS(α_max, Q_2)
10                          lb_AB ← p_2, ub_BC ← y − p_2
11                          IMPOSE-WAIT(⟨C, p''⟩)
12                          return STPPU J
```

Algorithm ODC-Unordered returns false if the network is not ODC; otherwise it returns the minimal network, as proved in the following theorem.

Theorem 10. *STPPU J, returned by algorithm ODC-Unordered, is minimal.*

To summarise, if the network passes all the tests, at the end we have the original unchanged constraint on AC, the new interval $[p_2, q'']$ for AB with its original preference function restricted to this interval, and the new interval $[y - p_2, x - q'']$ for BC with the corresponding preference function. We also have a wait of $\langle C, p'' \rangle$ on the AB constraint.

The wait means that, as in STPUs, the Unordered case is disjunctive: B must wait for C to occur *or* for p'' after A. Of course, if $x \geq p''$ then the wait will

always expire before C can occur. The lower bound of AB can be raised to p''; the disjunction is resolved. Consistently with [6] we will call this the *unconditional Unordered* reduction. Whether or not $x \geq p''$, observe that the lower bound of AB can be raised to x as in the *general Unordered* reduction [6].

Finally, then, we come to the true conditional case, when $x < p''$. First, if C occurs before p'' instants after A, we must take into account sub-problem Q_2. The fact that running Soft-PC-2 on Q_2 produced no changes on AC guarantees that there is a consistent and optimal choice for B. Second, if instead C occurs later than p'' after A, the fact that running Soft-PC-2 on Q_1 produced no changes on AC guarantees that there is a consistent and optimal choice for B. In particular, if $T_C \geq T_A + p''$, then B can occur at any point between p'' and q'' after A, with no restriction on whether it should follow or precede C.

We conclude that in the *conditional Unordered* case there is no way a priori to know which branch of the disjunction will hold; the wait is required. The straightforward method to resolve a wait is to branch on the two disjuncts. However, this will lead to a search of exponential complexity in a general network.[7]

7 Optimal Dynamic Controllability of General STPPUs

The reductions we have proposed give a procedure for checking ODC of triangular STPPUs. We have seen that, as in the case of hard constraints, the Unordered case may be solved by branching. However, we can consider just the unconditional reductions we have proposed and extend naturally algorithm 3DC [6] to 3ODCP (for *3-Optimal Dynamic Controllability with Preferences*).

In 3ODCP, for a general STPPU, we enforce the reductions explained above on all triangles that contain at least one contingent link. Triangles containing two contingent constraints are considered twice, with one link considered contingent and the other requirement, in turn.[8] The changes, on both preferences and interval ranges, are propagated to neighbouring triangles, until quiescence.

Just like 3DC, 3ODCP is sound but incomplete, i.e. if it fails then the STPPU P is not ODC, but it may succeed when P is not ODC. The example in [6] serves as proof of incompleteness also for 3ODCP, since any STPU can be mapped into an equivalent STPPU by adding to each constraint the constant $\mathbf{1}$ as preference function. Indeed, 3ODCP will fail whenever 3DC would have failed on the corresponding STPU obtained stripping the preference functions; but it will fail also due to changes on the preferences. There are two events regarding preferences that allow us to conclude that the entire STPPU is not ODC: the first event is the lowering of the preference of an element originally mapped to *opt*; the second event is the lowering of a preference α, at any level on any contingent link, associated with an element that appears in a solution with global preference at least α. Like 3DC, it can be proved that the complexity of 3ODCP is polynomial, but that it does not necessarily compute the minimal network.

[7] For general STPUs, 3DC+ ensures completeness by performing constraint propagation on waits, avoiding exponential search. This is future work for STPPUs.

[8] For technical reasons, triangles with three contingent links are prohibited [5].

8 Future Work

Temporal constraint problems in the real-world feature both preferences and uncertainty. In this paper we have introduced the STPPU and defined three levels of controllability. We have provided algorithms to determine whether the different levels hold, and shown that the complexity of checking controllability in a STPPU is the same as that for the equivalent notion in a STPU. In particular, the key notion of dynamic controllability can be tractably extended to account for preferences. We are currently working on the implementation and testing of the algorithms.

We plan to develop a complete algorithm (the analogue of 3DC+) for checking ODC in a STPPU, and then an algorithm for executing such a STPPU in a consistent, optimally dynamically controllable way. We are also investigating the use of probabilities over contingent constraints and their combination with preferences and uncertainty.

Acknowledgements. We thank Robert Morris and Carmen Gervet for discussions on STPPUs, and the reviewers for their constructive comments. The last author is partially supported by the EPSRC under grant GR/N64373/01.

References

1. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint solving and optimization. *Journal of the ACM*, 44(2):201–236, 1997.
2. R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.
3. L. Khatib, P. Morris, R. Morris, and K. B. Venable. Tractable Pareto optimization of temporal problems. In *Proc. of IJCAI'03*, pages 1289–1294, 2003.
4. L. Khatib, P. Morris, R. A. Morris, and F. Rossi. Temporal constraint reasoning with preferences. In *Proc. of IJCAI'01*, pages 322–327, 2001.
5. P. Morris and N. Muscettola. Execution of temporal plans with uncertainty. In *Proc. of AAAI-2000*, pages 491–496, 2000.
6. P. Morris, N. Muscettola, and T. Vidal. Dynamic control of plans with temporal uncertainty. In *Proc. of IJCAI'01*, pages 494–502, 2001.
7. C. Plaunt, A. Jónsson, and J. Frank. Run-time satellite telecommunications call handling as dynamic constraint satisfaction. In *Proc. of the 20th IEEE Aerospace Conference*, 1999.
8. F. Rossi, A. Sperduti, K. B. Venable, L. Khatib, P. Morris, and R. A. Morris. Learning and solving soft temporal constraints: An experimental study. In *Proc. of CP'02*, pages 249–263, 2002.
9. F. Rossi, K. B. Venable, L. Khatib, P. Morris, and R. Morris. Two solvers for tractable temporal constraints with preferences. In *Proc. of AAAI-02 Workshop on Preference in AI and CP*, 2002.
10. T. Vidal and H. Fargier. Handling contingency in temporal constraint networks: From consistency to controllabilities. *Journal of Experimental and Theoretical Artificial Intelligence*, 11(1):23–45, 1999.
11. T. Vidal and M. Ghallab. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *Proc. of ECAI-96*, pages 48–52, 1996.