# 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

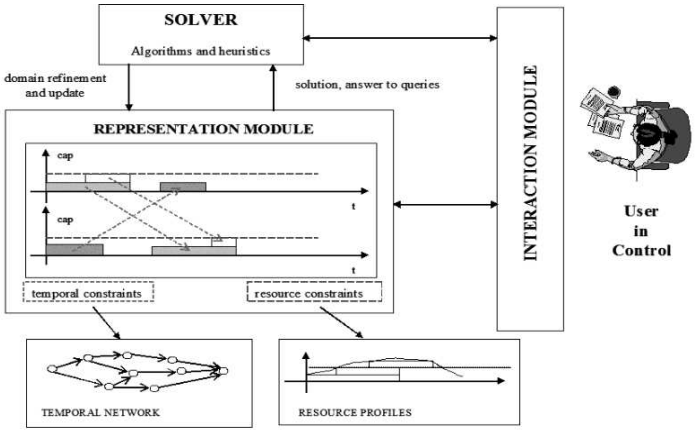


**Fig. 1.** The O-OSCAR Architecture.

**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.

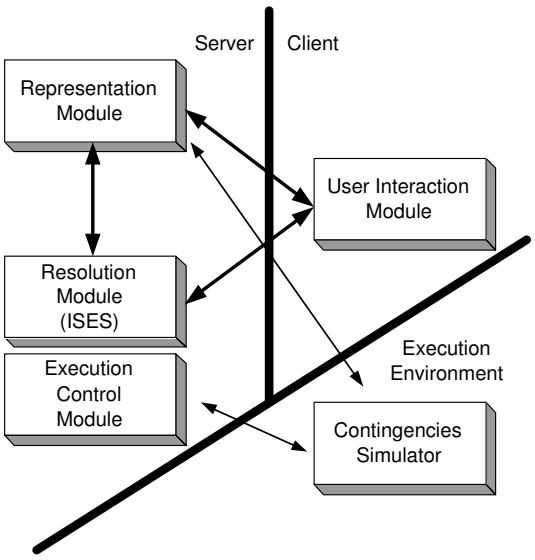


**Fig. 2.** The O-OSCAR's structure.

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 $\Delta$T in the activity duration is positive, then resource consistency will again have to be checked, exactly as in the previous case; in case $\Delta$T 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 ⟵ schedule_0
 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.