Reactive Recovery from Machine Breakdown in Production Scheduling with Temporal Distance and Resource Constraints

Roman Barták and Marek Vlk
Charles University in Prague, Faculty of Mathematics and Physics, Malostranské nám. 25, 118 00 Praha 1, Czech Republic

Keywords: Schedule Updates, Rescheduling, Predictive-reactive Scheduling, Constraint Satisfaction, Resource Failure.

Abstract: One of the classical problems of real-life production scheduling is dynamics of manufacturing environments with new production demands coming and breaking machines during the schedule execution. Simple rescheduling from scratch in response to unexpected events occurring on the shop floor may require excessive computation time. Moreover, the recovered schedule may be deviated prohibitively from the ongoing schedule. This paper studies two methods how to modify a schedule in response to a resource failure: right-shift of affected activities and simple temporal network recovery. The importance is put on the speed of the rescheduling procedures as well as on the minimum deviation from the original schedule. The scheduling model is motivated by the FlowOpt project, which is based on Temporal Networks with Alternatives and supports simple temporal constraints between the activities.

1 INTRODUCTION

Scheduling is a decision-making process of which the aim is to allocate limited resources to activities so as to optimize certain objectives. In manufacturing environment, developing a detailed schedule of the activities to be performed helps maintain efficiency and control of operations.

In the real world, however, manufacturing systems face uncertainty due to unexpected events occurring on the shop floor. Machines break down, operations take longer than anticipated, personnel do not perform as expected, urgent orders arrive, others are cancelled, etc. These disturbances render the ongoing schedule infeasible. In such case, a simple approach is to collect the data from the shop floor when the disruption occurs and to generate a new schedule from scratch. Gathering the information and total rescheduling involve excessive amount of time which may lead to failure of the scheduling mechanism and thus have far-reaching consequences.

For these reasons, reactive scheduling, which may be understood as the continuous correction of precomputed predictive schedules, is becoming more and more important. On the one hand, reactive scheduling has certain things in common with some predictive scheduling approaches, such as iterative improvement of some initial schedule. On the other hand, the major difference between reactive and predictive scheduling is the on-line nature and associated real-time execution requirements. The schedule update must be accomplished before the running schedule becomes invalid, and this time window may be very small in a complex manufacturing environment.

In this work we take the scheduling model from the FlowOpt project (Barták et al., 2012). Simply said, a schedule consists of activities, resources and constraints. Activities require resources to process them and all resources may perform at most one activity at a time. Possible positions of activities in time are restricted by simple temporal constraints.

The aim of this work is to propose a technique to recover a schedule from machine breakdown. The intention is to find a feasible schedule as similar to the original one as possible, and as fast as possible. The paper proposes two methods. The Right Shift Affected algorithm reallocates activities from the failed resource to available resources and then it keeps repairing violated constraints until the feasible schedule is obtained. The STN-Recovery algorithm retracts a certain subset of activities from resources and then it allocates one activity after another in suitable order in such a way that no constraints are violated. The major innovation is support for simple temporal constraints (Dechter, Meiri and Pearl, 1991) rather than assuming precedence constraints only.

We first survey briefly the closely related works on which our approaches are based on. Section 3 then

Barták R. and Vlk M..
Reactive Recovery from Machine Breakdown in Production Scheduling with Temporal Distance and Resource Constraints.
DOI: 10.5220/0005215701190130
In *Proceedings of the International Conference on Agents and Artificial Intelligence* (ICAART-2015), pages 119-130
ISBN: 978-989-758-074-1
Copyright © 2015 SCITEPRESS (Science and Technology Publications, Lda.)

explains the problem tackled in this work. The suggested methods are described in sections 4 and 5. The experimental results are given in section 6, and the final part points out possible future work.

2 RELATED WORKS

The field of rescheduling (predictive-reactive scheduling) has been addressed in a number of works, as surveyed for instance in (Raheja and Subramaniam, 2002), (Vieira et al., 2003), and (Ouelhadj and Petrovic, 2009). However, the algorithms discussed in the scheduling literature deal with scheduling problems that do not consider temporal constraints (minimal and maximal time lags) but usually only precedences. To the best of our knowledge, there is no algorithm that could be straightforwardly used for the problem with simple temporal constraints studied in this paper. Hence, we suggest to exploit and to integrate some known techniques to tackle this type of problem.

The fundamental inspiration comes from *heuristic-based* approaches, which do not guarantee to find an optimal solution, but respond in a short time. The simplest schedule repair technique is the *right shift rescheduling* (Abumaizar et al., 1997). This technique shifts the operations globally to the right on the time axis in order to cope with disruptions. When it arises from machine breakdown, the method introduces gaps in the schedule, during which the machines are idle. It is obvious that this approach results in schedules of bad quality, and can be used only for environments involving minor disruptions.

The shortcomings of total rescheduling and right shift rescheduling gave rise to another approach: *affected operation rescheduling*, also referred to as partial schedule repair (Smith, 1994). The idea of this algorithm is to reschedule only the operations directly and indirectly affected by the disruption in order to minimize the deviation from the initial schedule.

The *Repair-DTP* algorithm proposed in (Skalický, 2011) tackles a problem very similar to ours, however, it is designed to correct violated constraints in manually edited schedules. The model involves precedence constraints and synchronization constraints, but excludes minimum and maximum time lags. Nonetheless, in order to reduce searching space, the Repair-DTP algorithm employs *Simple Temporal Networks* (STN) (Dechter, Meiri and Pearl, 1991) and *Incremental Full Path Consistency* (IFPC) algorithm (Planken, 2008), which incrementally maintains the *All Pairs Shortest Path* (APSP) property. If a feasible correction exists, the algorithm tries to find the most similar schedule to the initial one through only shifting activities in time. Since the Repair-DTP algorithm does not try changes in resource selection, it cannot be used to deal with machine failure. Moreover, the main shortcoming of the algorithm is searching through disjunctions, introduced by hierarchical nature of the model and by resource unarity. This leads to excessive (exponentially growing) amount of temporal networks that are inspected, which requires unacceptable amount of time.

In the methods proposed further, apart from STN and IFPC algorithm, some widely used search techniques from the field of *Constraint Satisfaction* (Brailsford, Potts and Smith, 1999) are employed, namely *Conflict-Directed Backjumping with Backmarking* (Kondrak and Beek, 1997).

3 PROBLEM DEFINITION

3.1 Scheduling Problem

Scheduling problem P is a triplet of three sets: $Activities$, $Constraints$, and $Resources$.

- $Activities = \{$all activities in $P\}$
- $Constraints = \{$all temporal constraints in $P\}$
- $Resources = \{$all available resources in $P\}$

Each activity A is specified by its start time $Start(A)$ and end time $End(A)$, which we will look for, and fixed duration $Duration(A)$, which is part of the problem specification. All these numbers are nonnegative integers. Since we do not allow preemptions (interruptibility of activities), $Start(A) + Duration(A) = End(A)$ holds.

Temporal Constraints

Constraints determine mutual position in time of two distinct activities. Constraint $C \in Constraints$ is a triplet (A_i, A_j, w), where $A_i, A_j \in Activities$, $w \in \mathbb{Z}$, and the semantics is following.

$$Start(A_j) - Start(A_i) \leq w \quad (1)$$

Now, some terminology from the graph theory deserves to be clarified in terms of the scheduling model. Activities A_i and A_j are called *adjacent* if there exists a constraint (A_i, A_j, w) or (A_j, A_i, w) for any $w \in \mathbb{Z}$.

Two activities A_i and A_j are *connected* if there exists a sequence of activities $A_i, A_{i+1}, \ldots, A_{j-1}, A_j$ such that A_i and A_{i+1} are adjacent, A_{i+1} and A_{i+2} are adjacent, ..., A_{j-1} and A_j are adjacent. A *connected component* is a maximal (in terms of inclusion) subset of

activities such that all activities from the subset are connected. Each activity as well as each constraint belongs to exactly one connected component.

Resource Constraints

Let $A \in Activities$, then the set of resources that may process activity A is denoted $Resources(A)$. The set $Resources(A)$ is often referred to as a resource group.

Each activity needs to be allocated to exactly one resource from its resource group. Let $A \in Activities$, then a resource $R \in Resources(A)$ is *selected* if resource R is scheduled to process activity A, which we denote $SelectedResource(A) = R$.

Each activity must have a selected resource to make a schedule feasible. Formally:

$$\forall A \in Activities : SelectedResource(A) \neq null$$

All resources in a schedule are unary, which means that they cannot execute more activities simultaneously. Therefore, in a feasible schedule for all activities $A_i \neq A_j$ the following holds.

$$SelectedResource(A_i) = SelectedResource(A_j) \Rightarrow End(A_i) \leq Start(A_j) \vee End(A_j) \leq Start(A_i) \quad (2)$$

A Special Case

Real-life scheduling problems are usually designed in such a way that there are subsets of resources that share certain capabilities and which then constitute resource groups of activities. This observation may make some models easier to solve.

The resource groups of a scheduling problem are *equivalent* if one and only one of the following conditions holds for any two resource groups $Resources(A_1)$ and $Resources(A_2)$ of two distinct activities A_1 and A_2.

- $Resources(A_1)$ is equal to $Resources(A_2)$ $(Resources(A_1) = Resources(A_2))$
- $Resources(A_1)$ and $Resources(A_2)$ do not overlap $(Resources(A_1) \cap Resources(A_2) = \emptyset)$

If the resource groups are not equivalent, they are called *arbitrary*.

Motivated by the nature of real-life scheduling problems and their need for speed, the proposed algorithms anticipate that the resource groups are equivalent.

3.2 Schedule

A schedule S (sometimes referred to as a resulting schedule or a solution) is acquired by allocating activities in time and on resources. Allocation of activities in time means assigning particular values to the variables $Start(A)$ for each $A \in Activities$. Allocation of activities on resources means selecting a particular resource ($SelectedResource(A)$) from the resource group ($Resources(A)$) of each activity $A \in Activities$.

To make a schedule *feasible*, the allocation must be conducted in such a way that all the temporal constraints (1) as well as all the resource constraints (2) in the model are satisfied.

3.3 Rescheduling Problem

The problem we generally deal with is that we are given a particular instance of the scheduling problem along with a feasible schedule, and also with a change in the problem specification. The aim is to find another schedule that is feasible in terms of the new problem definition. The feasible schedule we are given is referred to as an original schedule or an ongoing schedule.

The machine breakdown, which is also referred to as a machine or resource failure, may happen in the manufacturing system at any point in time, say t_f, and means that a particular resource cannot be used anymore, i.e., for all $t \geq t_f$. This makes further questions arise, e.g., whether the activities that were being processed at time t_f are devastated and thus must be performed from the beginning, whether their predecessors must be also re-executed if there are only solutions violating temporal constraints, and many others.

For the sake of simplicity, let us assume that a resource fails at the beginning of the time horizon (at time point $t = 0$), i.e., right before the schedule execution begins. The resource that fails is in what follows also referred to as a forbidden resource. Formally, let S_0 be the schedule to be executed and R_f be the failed resource; the aim is to find a feasible schedule S_1, such that R_f is not used at any point in time $t \geq 0$. S_1 is referred to as a recovered schedule. The intention is to find S_1 as fast as possible and, regardless of the initial objectives, the more similar to S_0, the better. For this purpose we need to evaluate the modification distance.

Let us denote $Start_S(A)$ the start time of activity A in schedule S. In what follows we distinguish the following distance functions.

$$f_1 = \sum_{A \in Activities} |Start_{S_1}(A) - Start_{S_0}(A)|$$

$$f_2 = |\{A \in Activities \mid Start_{S_1}(A) \neq Start_{S_0}(A)\}|$$

$$f_3 = \max_{A \in Activities} |Start_{S_1}(A) - Start_{S_0}(A)|$$

4 RIGHT SHIFT AFFECTED

The Right Shift Affected algorithm is a greedy algorithm to tackle the machine breakdown disruption. For each $A \in Activities$, it is assumed throughout that the forbidden resource is deleted from the resource group of activity A, i.e., $Resources(A) = Resources(A) \setminus \{ForbiddenResource\}$.

The algorithm is aimed at moving as few activities as possible, i.e., optimizing the distance function f_2. The idea is to reallocate activities from the forbidden resource and then keep reallocating activities that violate some constraint until the schedule is feasible.

How to move (reallocate) the activities, how to repair the constraints, and in what order to pick the activities to repair the constraints is described next.

4.1 Reallocating Activities

Activities are reallocated as follows. Suppose the algorithm wants to repair a constraint in such a way that an activity A should be reallocated to a time point t. The natural idea was to reallocate the activity A exactly to the time point t even if there is no resource available for the required $Duration(A)$. Then, when a repair function verifies constraints, it would have to verify the resource constraints too and then repair according to the resource constraint violation. Unfortunately, there always turned out to be a model for which this method gets stuck in an infinite loop, regardless of the way the constraints are repaired and the sequence of activities to be repaired.

Consequently, the algorithm always allocates activity A in such a way that it does not violate any resource constraint. This is achieved through seeking a time point t^* (which is greater or equal to time point t) where activity A can be allocated without violating the resource constraints. Formally, when the algorithm desires to allocate activity A to time point t, then activity A is allocated to time point t^*, such that $t^* \geq t$ and $\forall t' : t' \geq t \wedge t' < t^*$ activity A cannot be allocated in t' without overlapping some other activity on any resource from $Resources(A)$.

Checking Resource Availability

In order to express whether or not a resource is free at a specified time interval, let us first define $Impedimentary(A,R,t)$ as the set of activities that preclude activity A from being allocated on resource R at time t.

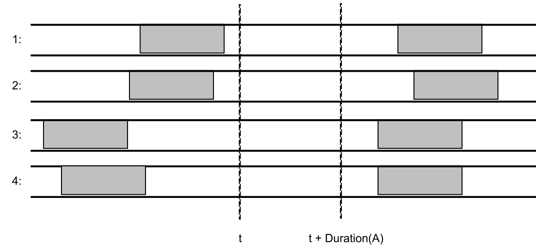

Figure 1: Illustration for ESSLPE rule.

$$Impedimentary(A,R,t) = \{A' \mid A' \in Activities \wedge R = SelectedResource(A') \wedge (t < End(A') \leq t + Duration(A) \vee t \leq Start(A') < t + Duration(A))\}$$

Now we can define a set of resources where activity A can be allocated at time t as such:

$$AvailableResources(A,t) = \{R \mid R \in Resources(A) \wedge Impedimentary(A,R,t) = \emptyset\}$$

Another question is which resource the algorithm should select if there are more resources available. Since the resource groups in the model are expected to be equivalent, it seems useful to pick the resource on which the activity best fits in terms of surrounding gaps. Therefore, the following heuristic is used.

Earliest Succeeding Start Latest Previous End (ESSLPE) Rule

Suppose activity A is about to be allocated at time t (see figure 1). The algorithm picks the resource with the earliest (closest) occupied time after the time point $t + Duration(A)$ (= earliest succeeding start), which holds for the resources number 3 and 4 in the figure 1. Like in this case, when there are more resources with the same earliest succeeding start, then the algorithm picks the resource with the latest (closest) occupied time before the time point t (= latest previous end), which is met by the resource number 4 in the figure 1. (If there are still ties, they may be broken arbitrarily.) Consequently, a resource that has at least some activity to process is always preferred to an empty resource.

Reallocation

The procedure `ReallocateActivity` (see algorithm 1) obtains two parameters: an activity to allocate (A) and a time point where it is desired to allocate the activity (t). Seeking for an available resource starts at time t, but the activity is ultimately allocated to the time point t^*, where an available resource is found.

Algorithm 1: Reallocating an activity.

function REALLOCATEACTIVITY(Activity A, TimePoint t)
  $SelectedResource(A) \leftarrow$ null
  $Start(A) \leftarrow$ null
  $t^* \leftarrow \min_{t' \geq t}\{t' \mid AvailableResources(A,t') \neq \emptyset\}$
  $Start(A) \leftarrow t^*$
  $SelectedResource(A) \leftarrow$ by ESSLPE rule from $AvailableResources(A,t^*)$
end function

4.2 Constraint Repair

The violated constraints are repaired as follows. When a temporal constraint between activities A_1 and A_2 of weight w is violated, it means that the distance between $Start(A_1)$ and $Start(A_2)$ is greater than allowed. Then the algorithm seeks for possible allocation of A_1 from the minimal time point that satisfies the constraint rightwards.

Here is where the title of the algorithm comes from. It repairs temporal constraints via moving activities to the right, which, of course, may cause violation of other temporal constraints. An important property is that when the algorithm picks an activity to be repaired, then it iterates over all temporal constraints associated with the activity being repaired until the activity does not violate any associated constraint.

Regardless of the order, in which the activities are selected to be repaired, the entire `RightShiftAffected` algorithm works as follows (see algorithm 2). First, it goes through all activities in the model and checks whether the activity uses the forbidden resource. In the positive case, the activity is reallocated through the `ReallocateActivity` procedure (seeking for available resources starts at the original start time of the activity), and the activity is added to the set $affected$. Now, none of the activities uses the forbidden resource and the set $affected$ contains activities that have been reallocated and therefore must be checked for temporal constraint violation.

Next, the algorithm takes an activity from the set $affected$ and proceeds to repair all violated temporal constraints associated with the activity in question. It repairs the constraints, as described, through moving activities to the right, so that if another activity is moved, it is added into the set $affected$ because it must be then checked for constraint violation. Recall that `ReallocateActivity` procedure always allocates an activity such that it does not violate any resource constraint, so that only temporal constraints are checked here. If the activity has been successfully healed, which means that the activity does not violate

Algorithm 2: Right Shift Affected.

function RIGHTSHIFTAFFECTED
  $affected \leftarrow \emptyset$
  for all $A \in Activities$ **do**
    if $SelectedResource(A) = ForbiddenResource$ **then**
      REALLOCATEACTIVITY($A, Start(A)$)
      $affected \leftarrow affected \cup \{A\}$
    end if
  end for
  while $affected \neq \emptyset$ **do**
    $A \leftarrow PopFrom(affected)$
    while $(A_1, A_2, w) \in ViolatedConstraints(A)$ **do**
      REALLOCATEACTIVITY(A_1, $Start(A_2) - w$)
      if $A_1 \neq A$ **then**
        $affected \leftarrow affected \cup \{A_1\}$
      end if
    end while
  end while
end function

any constraint, the algorithm proceeds to another one from $affected$.

As far as the order of taking activities from $affected$ is concerned, the best heuristic with respect to all conceivable performance measures turned out to be picking the rightmost activity, i.e., the activity with the maximum $Start(A)$. The explanation is that shifting the rightmost activities rightwards makes consecutively free space for shifting the activities allocated more on the left, which would otherwise have to creep over one another.

Termination

The algorithm successfully found a feasible schedule recovery for all input models that were assuredly solvable (which is guaranteed when there are more resources in each resource group than the number of activities in one connected component). However, the question whether the algorithm always ends and finds the solution, provided the schedule is recoverable, is still open.

If there is no feasible schedule recovery, the algorithm keeps repairing and never terminates. This is obviously the main shortcoming of the algorithm. One possible way to detect unrecoverability of the schedule is by passing and checking a time limit. Another way is to check where an activity is being allocated, and if the activity is allocated at a time point exceeding a certain threshold, it may be considered as an unsuccessful finding of a schedule.

5 STN-RECOVERY

The STN-Recovery is a bit more sophisticated algo-

rithm to tackle the machine breakdown. This algorithm anticipates that moving a large number of activities by small time is preferable to moving activities a lot in time. The basic idea is to deallocate some set of already scheduled activities and then allocate them back again. This is what is now meant by reallocation.

The point of the algorithm is to allocate connected components one after another through Conflict-Directed Backjumping. The allocation of an activity is carried out such that the start time of an activity is continuously incremented until an available resource at that time is found, or until the maximal possible value of the start time (which is determined with respect to the start times of already allocated activities) is exceeded. In the former case the algorithm proceeds to allocate the next activity, in the latter case the algorithm goes back to reallocate some previous activity. Since this allocation process might involve excessive computational burden, it is useful to prune the search space based on the fact that a resource failure leads only to deterioration of the schedule in the original optimization objective. Moreover, the group of resources where the broken down resource belongs is now likely to make a bottleneck. This assumption is used in such a way that the activities are reallocated from the broken down resource to available resources and then the activities are shifted so as they do not overlap each another – thus the minimal potential start times for allocation are obtained – and then the reallocation process can begin.

Firstly, the skeleton of the algorithm is given, and next, its particular steps are described in more details.

5.1 Skeleton of STN-Recovery

The STN (including the global predecessor) with the APSP property is supposed to have already been computed from the temporal constraints in the model; the resource constraints are not involved in the STN. Recall that the APSP property of the STN provides us the two-dimensional array w, of which the values say that $Start(A_j) - Start(A_i) \leq w[i, j]$, where $A_i, A_j \in Activities$.

A sketch of the STN-Recovery algorithm decomposed into 6 steps follows.

1. Find activities allocated to the forbidden resource and change their resource selection from the forbidden resource to an available resource, picking the resource with the lowest usage. Now some activities allocated on the same resource may overlap.
2. In order to find out which activities should be reallocated, do the following. For each resource (to which some activity has been added in step 1) shift the activities that overlap (to the right) so as they do not overlap, and add them into the set $affected$. Include in $affected$ also activities that were not actually shifted but are allocated on the right of those shifted.
3. For the sake of pruning the search space of the forthcoming reallocation, add STN constraints between the global predecessor and each activity in $affected$ so as to enforce that they can only start at the time they are currently allocated or later.
4. For each activity A in $affected$, acquire the connected component the activity A belongs to, and for all activities in all acquired connected components compute the values from which the allocation of the activity in the last step will begin (= $MinStart$), which is the maximum of (i) its current start time and (ii) its minimal distance from the global predecessor resulting from the STN.
5. Deallocate all activities in all connected components acquired in step 4.
6. Take the leftmost (according to the $MinStart$ values) non-allocated component C and allocate all activities in C starting with its leftmost activity using Conflict-Directed Backjumping with Backmarking. The activities within a connected component are allocated in the increasing order of their $MinStart$ values. Repeat this step until all connected components are allocated.

The skeleton of the algorithm is depicted in algorithm 3.

Algorithm 3: STN-Recovery.

Require: The STN with the APSP property
  function STN-RECOVERY
    for all $A \in Activities$ **do**
      if $SelectedResource(A) = ForbiddenResource$ **then**
        SWAPFORBIDDENSELECTION(A)
      end if
    end for
    $affected \leftarrow$ SHIFTONRESOURCES
    for all $A_i \in affected$ **do**
      IFPC(i, 0, $-Start(A_i)$)
    end for
    $components \leftarrow$ ACQUIRECOMPONENTS($affected$)
    DEALLOCATECOMPONENTS($components$)
    while $components \neq \emptyset$ **do**
      $C \leftarrow$ GETLEFTMOSTCOMPONENT($components$)
      ALLOCATECOMPONENT(C)
      $components \leftarrow components \setminus \{C\}$
    end while
  end function

5.2 Swapping Resource Selections

In the first step, the algorithm goes through all activities in the model and checks whether the activity is scheduled to be processed on the forbidden resource. In the positive case, the function `SwapForbiddenSelection(Activity A)` changes resource selection of activity A to some allowed resource.

It is not important which resource is selected because the activity is most likely going to be reallocated in the later steps. Nevertheless, the algorithm picks the resource with the lowest usage, which is the sum of the durations of the activities that are allocated to the resource in question.

Formally, let us first denote the set of activities that use resource R as such.

$$ResourceActivities(R) = \{A \in Activities \mid SelectedResource(A) = R\}$$

The usage of resource R can be written as follows.

$$Usage(R) = \sum_{A \in ResourceActivities(R)} Duration(A)$$

Then picking the resource with the lowest usage means this:

$$SelectedResource(A) = \arg \min_{R \in Resources(A)} (Usage(R))$$

At this time being, some activities may violate resource constraints.

5.3 Shifting Activities

In the second step, the algorithm repairs the violated resource constraints. It visits the resources one after another and shifts activities that overlap to the right. Since the original schedule is supposed to have been feasible, only the resources where some activities were added should be revised.

Procedure `ShiftOnResources` sweeps over the activities and conducts the shifting as follows. If activity A_0 overlaps activity A_1 on a resource, the activity with the later start time, say A_1, is set its start time to the end time of A_0. This shift may cause activity A_1 to overlap next activity, which is then set to start at the end of activity A_1 and so forth. The order of activities on the resource is preserved. All activities from the first activity that has been shifted up to the last activity (in terms of start times), even if some have not been shifted, are added to the set $affected$.

Formally, let $begin(R)$ be the start time of the first (earliest) activity that overlaps with another activity on resource R.

$$begin(R) \leftarrow \min_{A \in ResourceActivities(R)} \{Start(A) \mid \exists B \in ResourceActivities(R), B \neq A, Start(A) \leq Start(B) < End(A)\}$$

Further, let us denote R^i the i-th earliest activity allocated on resource R, which means that the following holds.

$$1 \leq i < j \leq |ResourceActivities(R)| \Rightarrow Start(R^i) \leq Start(R^j)$$

The activities on resource R are consecutively (from the leftmost activity) shifted such that:

$$Start(R^i) \leftarrow \max\{Start(R^i), End(R^{i-1})\}$$

Finally, the activities are added to the set $affected$ as follows.

$$affected \leftarrow \{A \in Activities \mid Start(A) > begin(SelectedResource(A))\}$$

This shifting may violate a large number of temporal constraints. The activities in the set $affected$ are going to be reallocated in the forthcoming steps. The reason why the set $affected$ includes the activities that have not been shifted, but are allocated on the right of the shifted activities, is, that they would otherwise preclude other activities from allocation.

5.4 Updating STN

In this step, the constraints determining the minimal distance of an activity from the global predecessor are added to the STN so as to modify the $MinStart$ values of activities to be reallocated, according to the start time values set in the previous shifting step. The IFPC algorithm is used because modifying the minimal start time of an activity affects the minimal start times of other activities from the same connected component.

Precisely, for each $A_i \in affected$, add to the STN via IFPC algorithm the constraint $(A_i, A_0, -Start(A_i))$, where A_0 denotes the global predecessor.

The point of adding this constraints is to reasonably maintain similarity to the original schedule, along with adequate pruning of the search space of the upcoming reallocation process.

5.5 Components Acquirement

There is still a question which and in what order the activities should be reallocated. Because shifting one activity is likely to violate temporal constraints

emanating from or to the activity, it is necessary to reallocate the entire connected component. Therefore, procedure `AcquireComponents(affected)` acquires the connected component that each activity $A \in affected$ belongs to, and the acquired connected component is added to the set $components$. After this step, $components = \{C_1, ...C_k\}$, where C_z for $z = 1, ..., k$ is a connected component.

In addition, for each activity, the *MinStart* value, which is the maximum of the current start time and of the minimal potential start time following from the STN (computed via IFPC in the previous step), is computed. Precisely, for each $C_z \in components$ and for each $A_i \in C_z$, assign:

$$MinStart(A_i) = Max\{Start(A_i), -w[i, 0]\}$$

As to the order for upcoming allocation, it is suitable to allocate activities in the increasing order of the *MinStart* values. The activity in a connected component with the lowest *MinStart* value is referred to as the leftmost activity. The leftmost connected component is the connected component of which the leftmost activity has the lowest *MinStart* value among all connected components. The algorithm always selects for allocation the leftmost component that has not yet been allocated.

5.6 Deallocation

Since the best way for allocating activities turned out to be the way without violating resource constraints, it is necessary to deallocate all activities in the connected components acquired in the previous step. Otherwise they would preclude other activities from allocation. Procedure `DeallocateComponent(components)` deallocates activities from each connected component $C \in components$, which means that for each $A \in C$: $Start(A) = null$ and $SelectedResource(A) = null$. After this (fifth) step, all activities from $components$ are deallocated.

5.7 Allocation

Allocating an activity again means searching for the time point when there is an available resource for the required duration. The resources are selected according to the ESSLPE rule described in 4.1.

In order to allocate a connected component, Conflict-Directed Backjumping with Backmarking is used (see algorithm 4). When an activity cannot be successfully allocated, it is necessary to jump back to the activity that is causing the conflict. For keeping the information which activity is conflicting with the activity being allocated, the conflict set for each activity is remembered. For this purpose, $cs[i]$ is a set of activities conflicting with A_i.

The activities are going to be allocated in the increasing order of their indexes that are determined according to their *MinStart* values. Thus we can anticipate that the connected component to be allocated, which is passed as a parameter, consists of activities $A_1, ..., A_n$. When two activities are compared, i.e., $A_j < A_i$, it means that their indexes are compared ($j < i$).

There are two possible causes why an activity cannot be allocated: a temporal conflict and a resource conflict.

Temporal Conflicts

Temporal conflicts are handled in procedure `UpdateBounds(Activity A)` (see algorithm 5), which is called before activity A_i is going to be allocated (line 6). In this procedure, the bounds of possible time allocation for activity A_i are computed according to the STN and start times of already allocated activities.

The lower bound of an activity is initially set to the *MinStart* value acquired in the previous steps. Then the procedure goes through the already allocated activities within the connected component in the same order as they have been allocated and updates bounds of A_i. Precisely, for each $k < i$, if $Start(A_k)$ + "minimal distance from $Start(A_k)$ to $Start(A_i)$" is greater than the current lower bound, then increase the lower bound, and add A_k to the conflict set of A_i. Similarly, if $Start(A_k)$ + "maximal distance from $Start(A_k)$ to $Start(A_i)$" is smaller than the current upper bound, then decrease the upper bound, and add A_k to the conflict set of A_i. The reason why activity A_k is added to the conflict set is that changing the start time of A_k creates (straight away or after a number of steps) some new possible start time for A_i.

Resource Conflicts

As far as resource conflicts are concerned, recall that $Impedimentary(A_i, R, t)$, which has been formally introduced in section 4.1, is a set of activities that preclude activity A_i from selecting resource R at time t. To make it possible to allocate activity A_i on resource R at time t, all activities from the set $Impedimentary(A_i, R, t)$ would have to be reallocated. Hence, among the activities in $Impedimentary(A_i, R, t)$, the activity that has been the least recently allocated (from the connected component being allocated) is added to the conflict set of activity A_i. But if there is an activity in

Algorithm 4: Allocating entire connected component.

1: **function** ALLOCATECOMPONENT(Activities $A_1, ..., A_n$)
2: $i \leftarrow 1$
3: **while** $i \leq n$ **do**
4: $newVal \leftarrow newVals[i]$ ▷ initially 0
5: **if** $newVal = 0$ **then**
6: UPDATEBOUNDS(A_i)
7: $newVal \leftarrow LowerBound(A_i)$
8: **end if**
9: **while** $SelectedResource(A_i) = null$ & $newVal \leq UpperBound(A_i)$ **do**
10: **if** $newVal \in Keys(Mark[i])$ & $Max(Mark[i][newVal]) < BackTo[i][newVal]$ **then**
11: $cs[i] \leftarrow cs[i] \cup Mark[i][newVal]$
12: $newVal \leftarrow newVal + 1$
13: continue
14: **end if**
15: $BackTo[i][newVal] \leftarrow A_i$
16: $newConflicts \leftarrow \emptyset$
17: **for all** $R \in Resources(A_i)$ **do**
18: $newConflicts \leftarrow newConflicts \cup Min^*(Impedimentary(A_i, R, newVal))$
19: **end for**
20: **if** $AvailableResources(A_i, newVal) \neq \emptyset$ **then**
21: $SelectedResource(A_i) \leftarrow$ by ESSLPE rule from $AvailableResources(A_i, newVal)$
22: $Start(A_i) \leftarrow newVal$
23: ▷ $newVal$ can be tried again
24: $Keys(Mark[i]) \leftarrow Keys(Mark[i]) \setminus \{newVal\}$
25: **else**
26: $Keys(Mark[i]) \leftarrow Keys(Mark[i]) \cup \{newVal\}$
27: $Mark[i][newVal] \leftarrow newConflicts$
28: **end if**
29: $cs[i] \leftarrow cs[i] \cup newConflicts$
30: $newVal \leftarrow newVal + 1$
31: **end while**
32: **if** $SelectedResource(A_i) = null$ **then**
33: $A_j \leftarrow Max(cs[i])$
34: $cs[j] \leftarrow cs[j] \cup cs[i] \setminus \{A_j\}$
35: **for** $k \leftarrow j + 1$ **to** n **do**
36: **for all** $key \in Keys(BackTo[k])$ **do**
37: $BackTo[k][key] \leftarrow Min(BackTo[k][key], A_j)$
38: **end for**
39: **end for**
40: **while** $i > j$ **do** ▷ jump back to j
41: $newVals[i] \leftarrow 0$
42: $i \leftarrow i - 1$
43: $SelectedResource(A_i) \leftarrow null$
44: $Start(A_i) \leftarrow null$
45: **end while**
46: **else**
47: $newVals[i] \leftarrow newVal$
48: $i \leftarrow i + 1$
49: **end if**
50: **end while**
51: **end function**

Algorithm 5: Updating lower and upper bounds.

function UPDATEBOUNDS(Activitiy A_i)
$cs[i] \leftarrow \emptyset$ ▷ clear conflict set
$LowerBound(A_i) \leftarrow MinStart(A_i)$
$UpperBound(A_i) \leftarrow \infty$
for $k \leftarrow 1$ **to** $i - 1$ **do**
$newValue \leftarrow Start(A_k) - w[i,k]$
if $LowerBound(A_i) < newValue$ **then**
$LowerBound(A_i) \leftarrow newValue$
$cs[i] \leftarrow cs[i] \cup \{A_k\}$
end if
$newValue \leftarrow Start(A_k) + w[k,i]$
if $UpperBound(A_i) > newValue$ **then**
$UpperBound(A_i) \leftarrow newValue$
$cs[i] \leftarrow cs[i] \cup \{A_k\}$
end if
end for
end function

$Impedimentary(A_i, R, t)$ from another connected component, which means it cannot be deallocated, then no activity is added to the conflict set.

This is exactly what Min^* does (at line 18). Formally, let C be the connected component being allocated. If $Impedimentary(A_i, R, t) \subseteq C$, then:

$$Min^*(Impedimentary(A_i, R, t)) = \arg \min_{A_k \in Impedimentary(A_i, R, t)} \{k\}$$

Otherwise $Min^*(Impedimentary(A_i, R, t)) = \emptyset$.

For illustration, when the algorithm is allocating activity A_7 and there are activities A_2, A_4, and A_6 inhibiting on a resource, then activity A_2 is added to the conflict set. If there is an activity from different, already allocated component, then no activity is added to the conflict set.

Further, recall $AvailableResources(A_i, t)$ is a subset of available resources from which the resource according to the ESSLPE rule is selected. Regardless of the result of the search for an available resource, the conflicting activities are merged into the conflict set of the activity being allocated (line 29).

Backjump

When the algorithm is about to conduct a backjump (starting at line 32), which happens when all possible start times of A_i have been tried, the most recently allocated activity from the conflict set of A_i is found (line 33). Let us denote this activity as A_j. Next, before deallocating activities that are jumped over, the activities from the conflict set of A_i except activity A_j are added to the conflict set of A_j.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Backmarking

The backmarking technique is implemented as follows. Firstly, the time horizon is infinite so that the structures *BackTo* and *Mark* cannot be simple two-dimensional arrays but arrays of dictionaries. Precisely, *BackTo* is an array of size n, $BackTo[i]$ is a dictionary, where keys are the (attempted) start times of the activity, and values are activities, i.e., $BackTo[i][newVal]$ is the lowest-indexed activity whose instantiation has changed since activity A_i was last tried to be allocated at time $newVal$.

As to the structure *Mark*, there is one difference. Notice that when the algorithm cannot find an available resource for activity A_i at time $newVal$, not only one, but a number of activities may be added to the conflict set of A_i. Consequently $Mark[i][newVal]$ is a set of activities, of which at least one must be reallocated in order to make activity A_i allocatable at time $newVal$. Therefore, when values *BackTo* and *Mark* are to be compared, it is firstly checked, whether there is *newVal* among the keys of $Mark[i]$, and in the positive case, $Max(Mark[i][newVal])$ and $BackTo[i][newVal]$ are compared (see line 10).

If $Max(Mark[i][newVal]) < BackTo[i][newVal]$, it means that none of the conflicting activities has been re-instantiated and thus it makes no sense to look for an available resource. However, before proceeding to the next value of *newVal*, it is necessary to merge the conflicting activities to the current conflict set (line 11) as if the search for an available resource was conducted – this is the reason why $Mark[i][newVal]$ must store the set of activities (and not just the most recent activity).

Oppositely, if *newVal* is not presented among the keys of $Mark[i]$ or $Max(Mark[i][newVal]) \geq BackTo[i][newVal]$, the algorithm does look for an available resource. If activity A_i is successfully allocated, the key *newVal* is removed from $Mark[i]$ (line 24), otherwise $Mark[i][newVal]$ stores the conflicting activities (line 27).

Termination

Notice that the algorithm does not check for the recoverability of the disrupted ongoing schedule, which means that if there is no feasible solution, the procedure `AllocateComponent(Component C)` never terminates. This can be solved by giving it a limited time (cut-off limit), or by detecting that the method got stuck in a loop, which may be proven for example when it tries to allocate an activity in time greater than the maximal estimate of makespan (which may be the sum of the durations of all activities and of all minimal distances in the model).

6 EXPERIMENTAL RESULTS

The STN-Recovery algorithm is designed to move a lot of activities by a small amount of time, which means that it should not be used when minimizing the number of shifted activities (objective f_2). On the other hand, the algorithm should perform well in minimizing the biggest shift of an activity (objective f_3). On the contrary, the Right Shift Affected algorithm intents to affect only the necessary subset of activities, making it better when minimizing the objective f_2. Oppositely, if the alternative resources for the broken-down resource make a bottleneck, the affected activities (and subsequently all connected components with them) are moved to the end of the schedule horizon. This is expected to yield a poor performance in the objective f_3, which is unacceptable when the original schedule objective is related to lateness or tardiness. The distance functions f_2 and f_3 are expected to grow linearly with increasing number of activities in the model for both algorithms.

To support the above hypotheses we performed experiments with randomly generated problems composed of 20 resources in one group. Each connected component consists of 8 activities and up to 28 temporal constraints (some may be redundant). Having more resources in a group than the number of activities in a component ensures recoverability from a resource failure. We also included a total rescheduling algorithm (rescheduling from scratch) in the comparison to justify the claims from the introduction. The algorithms were running on Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz, 3701 Mhz, kernels: 4, logical processors: 8; RAM: 8,00 GB.

Briefly speaking, the experimental results confirmed the hypotheses. As depicted in figure 2, the Right Shift Affected algorithm is far better when optimizing the distance function f_2, but the STN-Recovery algorithm is significantly better when optimizing the distance function f_3, as shown in figure 3. As far as function f_1 is concerned (which is the total sum of shifts), the STN-Recovery algorithm outperforms the Right Shift Affected, but the difference is negligible.

The Right Shift Affected algorithm is somewhat faster than STN-Recovery (see figure 4), however, STN-Recovery has the following advantage. The algorithm always allocates the leftmost connected component that has not been allocated yet, therefore, when the algorithm is allocating the connected component with the leftmost activity that has the *MinStart* value t, the schedule is not going to be modified before time point t. This allows the system to keep executing an ongoing schedule even if it has not been

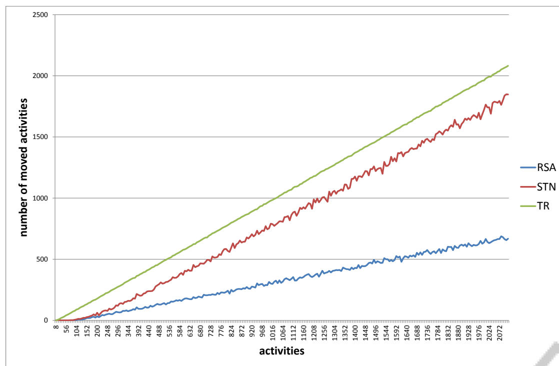

Figure 2: The number of shifted activities for Right Shift Affected, STN-Recovery, and Total Rescheduling.

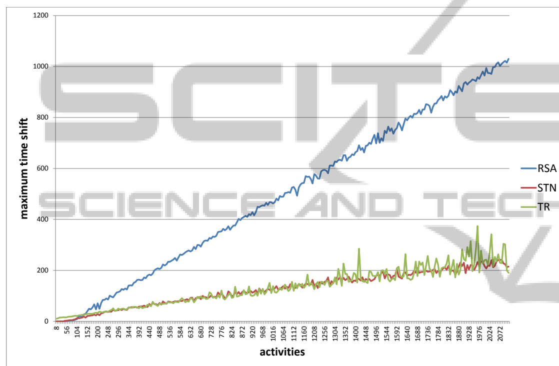

Figure 3: The biggest shift of an activity for Right Shift Affected, STN-Recovery, and Total Rescheduling.

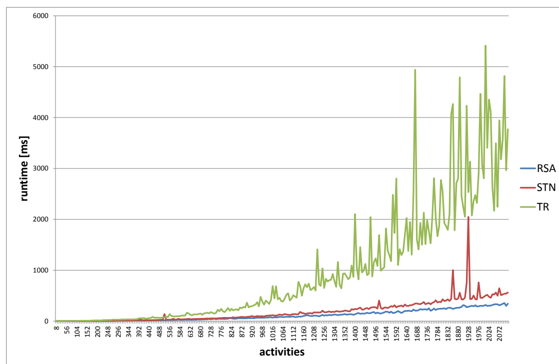

Figure 4: Run times for Right Shift Affected, STN-Recovery, and Total Rescheduling.

completely recovered yet.

The dependencies on the density of constraints showed no tendency. However, one might wonder how the algorithms perform as the size of connected components increases. As depicted in figure 5, there are alarmingly longer run-times of STN-Recovery for some models, but exponential growth is not apparent, unlike in the case of total rescheduling, which turned out to be useless by the component size of 33.

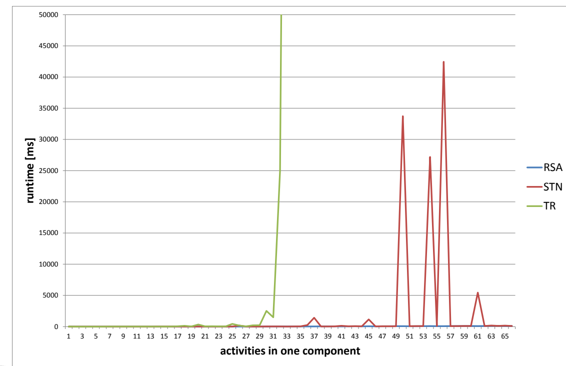

Figure 5: Run times for Right Shift Affected, STN-Recovery, and Total Rescheduling, dependent on the number of activities in one component.

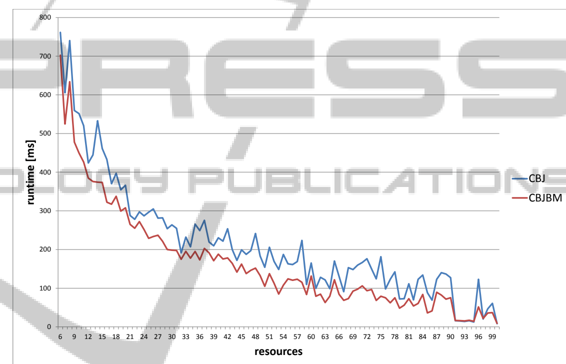

Figure 6: Run times for Conflict-Directed Backjumping and Conflict-Directed Backjumping with Backmarking.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

As far as the backmarking technique is concerned, it brought some saving of time as expected, because determining availability of a resource is carried out in logarithmic time in the number of activities on the resource. On the other hand, as the number of resources in the model decreases below a certain number, one might expect backmarking to become counterproductive owing to the overhead costs. Nevertheless, according to figure 6, backmarking pays regardless of the number of resources in the model.

7 CONCLUSIONS

This paper proposed two different methods to handle a resource failure, i.e., a disruption when a resource suddenly cannot be used anymore by any activity, which may occur during a schedule execution.

The first method takes the activities that were to be processed on a broken machine, reallocates them, and then it keeps repairing violated constraints until it gets a feasible schedule. This approach is suitable when

it is desired to move as few activities as possible; however, the question whether the algorithm always ends is still open. The second method deallocates a subset of activities and then it allocates them back through Conflict-Directed Backjumping with Backmarking. This approach is useful when the intention is to shift activities by a short time distance, regardless of the number of moved activities.

The main shortcoming is that if there is no feasible recovery of the ongoing schedule, neither of the methods is able to quickly and securely report it. In real-life environments, however, the schedule recoverability from the breakdown of any particular machine is often known (for instance the minimum required number of available resources of each resource group may be obvious) or can be computed before the schedule execution begins.

Both suggested algorithms may be easily adapted to handle the models with arbitrary resource groups, and also to cope with another disturbance – hot order arrival (Vlk, 2014).

Further investigation is needed for determining the conditions under which a schedule is recoverable. Next, it may be of interest to generalize the algorithms for models that involve for example interruptibility of activities, various speeds of resources, setup times of resources or calendars of availabilities of resources.

ACKNOWLEDGEMENTS

Research is supported by the Czech Science Foundation under the project P103/10/1287 and by the Charles University in Prague, project GA UK No. 178915.

REFERENCES

Abumaizar, R. J. - Svestka, J. A. (1997). *Rescheduling Job Shops under Random Disruptions.* International Journal of Production Research, 35(7), 2065-2082.

Barták, R. - Jaška, M. - Novák, L. - Rovenský, V. - Skalický, T. - Cully, M. - Sheahan, C. - Thanh-Tung, D. (2012). *FlowOpt: Bridging the Gap Between Optimization Technology and Manufacturing Planners.* Proceedings of ECAI 2012, pp. 1003-1004, IOS Press.

Brailsford, S.C. - Potts, Ch.N. - Smith, B. M. (1999). *Constraint satisfaction problems: Algorithms and applications.* European Journal of Operational Research 119, 557-581.

Dechter, R. - Meiri, I. - Pearl, J. (1991). *Temporal constraint networks.* Artificial Intelligence 49(1-3), 61–95.

Kondrak, G. - Beek, P. van (1997). *A Theoretical Evaluation of Selected Backtracking Algorithms.* Artificial Intelligence 89, 365-387.

Ouelhadj, D. - Petrovic, S. (2009). *A survey of dynamic scheduling in manufacturing systems.* Journal of Scheduling, v.12 n.4, p.417-431.

Planken, L. R. (2008). *New Algorithms for the Simple Temporal Problem.* Delft, the Netherlands, 75 p. Master's thesis, Delft University of Technology.

Raheja, A. S. - Subramaniam, V. (2002). *Reactive recovery of job shop schedules – a review.* International Journal of Advanced Manufacturing Technology, 19, 756-763.

Skalický, T. (2011). *Interactive Scheduling and Visualisation.* Prague, 95 p. Master's thesis, Charles University in Prague.

Smith, S.F. (1994). *Reactive Scheduling Systems.* In: D. Brown and W. Scherer (eds.), Intelligent Scheduling Systems.

Vieira, G. - Herrmann, J. - Lin, E. (2003). *Rescheduling manufacturing systems: a framework of strategies, policies, and methods.* Journal of Scheduling 6: 39-62, Kluwer Acad. Publishers.

Vlk, M. (2014). *Dynamic Scheduling.* Prague, 72 p. Master's thesis, Charles University in Prague.

SCIENCE AND TECHNOLOGY PUBLICATIONS