Hierarchical Task Model with Alternatives for Predictive-reactive Scheduling

Marek Vlk and Roman Barták (supervisor)
Charles University in Prague, Faculty of Mathematics and Physics
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
{vlk, bartak}@ktiml.mff.cuni.cz

Abstract

Attaining optimal results in real-life scheduling is hindered by a number of problems. One such problem is dynamics of scheduling environments with breaking-down resources and hot orders coming during the schedule execution. Traditional approach to react to unexpected events occurring on the shop floor is generating a new schedule from scratch. Complete rescheduling, however, may require excessive computation time. Moreover, the recovered schedule may deviate a lot from the ongoing schedule. Some work has focused on tackling these shortcomings, but none of the existing approaches tries to substitute jobs that cannot be executed with a set of alternative jobs. This paper describes the scheduling model suitable for dealing with unforeseen events using the possibility of alternative processes and states the future goals.

Introduction

Scheduling, the aim of which is to allocate scarce resources to activities in order to optimize certain objectives, has been frequently addressed in the past decades. Developing a detailed schedule in manufacturing environment helps maintain efficiency and control of operations.

In the real world, however, manufacturing systems face uncertainty owing to unforeseen 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 canceled, etc. These disturbances may bring inconsistencies into the ongoing schedule. If the ongoing schedule becomes infeasible, the simple approach is to collect the data from the shop floor when the disruption occurs and to generate a new schedule from scratch. Because most of the scheduling problems are NP-hard, complete rescheduling usually involves prohibitive computation time and an excessive deviation of the recovered schedule from the original schedule.

To avoid the problems of rescheduling from scratch, reactive scheduling, which may be conceived as the continuous correction of precomputed predictive schedules, is becoming more and more important. Reactive scheduling is contradistinguished from predictive scheduling mainly by its 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 short in complex scheduling environments.

Several novel sophisticated methods attempt to cope with the shortcomings of complete rescheduling, e.g., by rescheduling only the activities somehow affected by the disturbance. To the best of our knowledge, however, none of the existing approaches tries to replace some activities by a set of alternative activities (using other available resources) to achieve the same goal.

In this paper we propose a model suitable for development of algorithms for modifying a schedule to accommodate disturbances, such as a machine breakdown, using the possibility of alternative processes, i.e., to re-plan the influenced part of the schedule.

Related Work

The approaches how to tackle dynamics of the scheduling environment can be divided basically into two branches according to whether or not the predictive schedule is computed before the execution starts (Vieira, Herrmann, and Lin 2003). If the predictive schedule is not computed beforehand and individual activities are assigned to resources pursuant to some so called dispatching rules during the execution, we talk about *completely reactive scheduling* or on-line scheduling. This strategy is suitable for very dynamic environments, where it is not known in advance which activities it will be necessary to process. On the other hand, it is obvious that this approach hardly ever leads to an optimal or near-optimal schedule.

If the schedule is crafted beforehand and then updated during its execution, it is referred to as *predictive-reactive scheduling*. When correcting the ongoing schedule in response to changes within the environment, the aim is usually to minimize the schedule modification. The motivation for minimizing the alteration of the schedule is that every aberration may lead to deterioration in the performance owing to affecting other planning activities based upon the original schedule. Similarity of two schedules may be formally defined for example as a *minimal perturbation problem* (Barták, Muller, and Rudová 2003).

There is an extensive literature on rescheduling (Ouelhadj and Petrovic 2009; Raheja and Subramaniam 2002). First,

the *heuristic-based* approaches do not guarantee finding an optimal solution, but they respond in a short time. The simplest schedule-repair technique is the *right shift rescheduling* (Abumaizar and Svestka 1997), which shifts the operations globally to the right on the time axis in order to cope with disruptions. This may lead to schedules of very bad quality.

Another simple heuristic is *affected operation rescheduling* (Smith 1995), also referred to as partial schedule repair, the essence of which is to reschedule only the operations directly and indirectly affected by the disruption in order to minimize the deviation from the initial schedule.

Better schedules to the detriment of computational efficiency may be attained by using *meta-heuristics* such as simulated annealing, genetic algorithms, tabu search, and iterative flattening search (Oddi et al. 2007). These high level heuristics guide local search methods to escape from local optima by occasional accepting worse solutions or by generating better initial solutions for local probing in some sophisticated way.

Some techniques from the field of artificial intelligence and knowledge-based systems are also applied in rescheduling, namely *case-based reasoning* (Cunningham and Smyth 1997), *fuzzy logic* (Ramkumar, Tamilarasi, and Devi 2011), and *neural networks* (Jain and Meeran 1998). Another approach, which is rather an independent branch, is *multi-agent based architectures* (Zhang et al. 2011). Multi-agent systems seem to be the most promising approach, but the coordination among the agents is hard to achieve.

The attempts to absorb certain amount of uncertainty based on the past executions of schedules is considered in another strategy, usually referred to as *robust proactive scheduling*. One such example is a model and an algorithm generating a predictive schedule of production workflows that is (proactively) robust with regard to so called immediate events, which include breakdown of a workstation and faulty termination of a workflow execution (Dulai and Werner-Stark 2015). The robustness is attained by shifting activities (traditional introducing or enlarging gaps on resources) based on the probabilities of resource failures, which are estimated according to previous experiences. The drawback of the algorithm is the assumption that every resource failure is only temporary, and the time for how long the resource is unavailable in case of its failure is known in advance.

While there is a great amount of work devoted to planning with time and resources and to integrating planning and scheduling techniques, to the best of our knowledge, there is no research carried out aiming at the possibility of re-planning in the field of predictive-reactive scheduling.

Scheduling Model Description

The scheduling model we work with is taken from the FlowOpt system (Barták et al. 2012), which contains a tool for designing and editing *manufacturing workflows*. Workflow in general may be understood as a scheme of performing some complex process, itemized into simpler processes and relationships among them. Manufacturing workflow is then an outline how to obtain a desired product.

In order to make editing of workflows easier, the workflows in our model match up the structure of Nested Temporal Networks with Alternatives (Barták and Čepek 2007), where the nodes of a network correspond to the tasks of a workflow. The tasks decompose into other tasks, referred to as their children. There are two types of decomposition: parallel and alternative. The tasks that do not decompose further (i.e., leaves) are called *primitive tasks*. The primitive tasks correspond to activities (or operations) and are associated with some additional parameters, namely start, end, and duration.

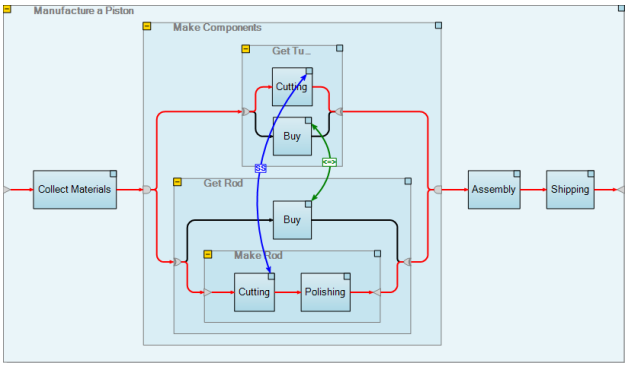

Figure 1: An example of a workflow (Skalický 2011).

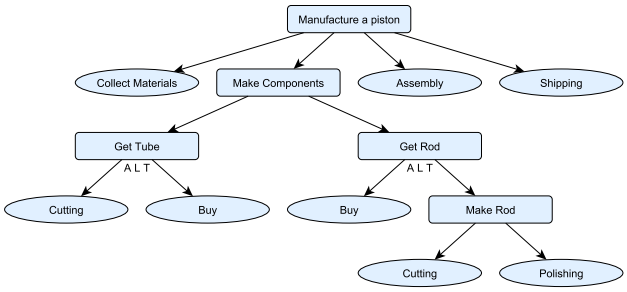

Figure 2: A decomposition tree for the workflow. The label "ALT" beneath tasks stands for alternative decomposition; the other decompositions are parallel. Activities are ellipse-shaped.

An example of a workflow and its decomposition tree are depicted in Figures 1 and 2. It contains eight primitive tasks (activities), three parallel tasks, and two alternative tasks.

The workflows as described define a number of feasible processes. A *process* is a subset of tasks selected to be processed. While a *parallel task* requires all its children to be processed, an *alternative task* requires exactly one of its children to be processed. If an arbitrary task is not in a process, none of its offspring is in the process either. Hence, to ensure that an instance of a workflow is actually processed, its root task has to be in the selected process. An example of a process is depicted in Figure 3.

To introduce some restrictions in terms of occurrences of tasks in the process and their time data, a pair of tasks can be bound by a *constraint*. Temporal constraints include *precedences* (one task has to be accomplished before the execution of another task starts), and *synchronizations* (one task has to

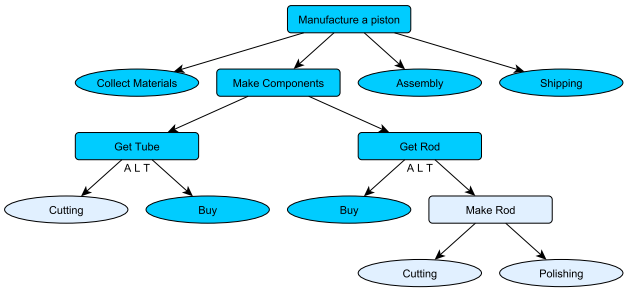

Figure 3: The decomposition tree with a selected process.

start/end exactly when another task starts/ends). Logical constraints include *implications* (if one task is in the process, the other task must be in the process too), *equivalences* (either both tasks must be in the process or none of them can be in the process), and *mutual exclusions* (at most one of the tasks can be in the process).

In Figure 1, there is one equivalence constraint enforcing that a tube and a rod are both either bought or made. The other constraint (synchronization) ensures that the activities "cutting tube" and "cutting rod" start at the same time.

Besides these constraints added by a user, which are referred to as custom constraints, there are some implicit constraints arising from the hierarchical structure of tasks. For example, the start time of a task equals the start time of its earliest child, and the end time of the task equals the end time of its latest child.

Activities are processed on *resources*. All resources are unary, which means that each resource may perform no more than one activity at a time. This limitation is often referred to as a resource constraint and belongs to the mentioned implicit constraints. Each activity is specified by a set of resources on which the activity can be processed (resource group), and in the resulting schedule, each activity in the selected process must be allocated to exactly one resource (selected resource).

Note that workflow is only a guideline how to manufacture a particular product. If a user wants n such products, n instances of the corresponding workflow are inserted into the model. An instance of a workflow is referred to as an *order*. It contains some additional data, such as due date and lateness penalty.

Finally, a resulting schedule is *feasible* if all custom as well as implicit constraints are satisfied.

Scheduling Problem

Formally, schedule S is a triplet of three sets: $Tasks$, $Constraints$, and $Resources$.

Tasks Tasks match up a forest structure. Therefore, every task T either has a parent, i.e., $\exists P \in Tasks : parent(T) = P$, or is a root, i.e., $parent(T) = null$.

- $subtasks(T) = \{C \in Tasks \mid parent(C) = T\}$
- $Roots = \{R \in Tasks \mid parent(R) = null\}$

There are three types of tasks: *parallel*, *alternative*, and *primitive* tasks.

- $Tasks = Parallel \cup Alternative \cup Primitive$
- $\forall T \in Parallel \cup Alternative : subtasks(T) \neq \emptyset$
- $\forall T \in Primitive : subtasks(T) = \emptyset$

Let process $P \subseteq Tasks$ be the set of tasks selected to be processed. Making the process feasible introduces the following constraints:

- $T \in P \cap Parallel : subtasks(T) \subseteq P$
- $T \in P \cap Alternative : |subtasks(T) \cap P| = 1$
- $T \notin P : subtasks(T) \cap P = \emptyset$

Let S_i and E_i denote the start time and end time, respectively, of task T_i. Each activity corresponding to the primitive task T_i is specified by the duration D_i. Then the time data are computed as follows:

- $\forall T_i \in P \cap Primitive : S_i + D_i = E_i$
- $\forall T_i \in P \cap (Parallel \cup Alternative) :$
  $S_i = \min\{S_j \mid T_j \in subtasks(T_i) \cap P\}$
  $E_i = \max\{E_j \mid T_j \in subtasks(T_i) \cap P\}$

Constraints There are two basic types of constraints: temporal, and logical. Temporal constraints restrict mutual position in time of two distinct activities. We take into consideration precedence and synchronization constraints, the semantics of which is as follows:

- $(i \rightarrow j) : T_i, T_j \in P \Rightarrow E_i \leq S_j$
- $(i\ ss\ j) : T_i, T_j \in P \Rightarrow S_i = S_j$
- $(i\ se\ j) : T_i, T_j \in P \Rightarrow S_i = E_j$
- $(i\ es\ j) : T_i, T_j \in P \Rightarrow E_i = S_j$
- $(i\ ee\ j) : T_i, T_j \in P \Rightarrow E_i = E_j$

Logical constraints are of three types: implications, equivalences, and mutexes. The semantics of the constraints is such:

- $(i \Rightarrow j) : T_i \in P \Rightarrow T_j \in P$
- $(i \Leftrightarrow j) : T_i \in P \Leftrightarrow T_j \in P$
- $(i\ mutex\ j) : T_i \notin P \vee T_j \notin P$

Resources Let $T \in Primitive$, then the set of resources that may process the primitive task T is denoted $Resources(T)$. The set $Resources(T)$ is often referred to as a resource group.

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

Each primitive task that is selected to the process P must have a selected resource to make a schedule feasible. Formally:

$$\forall T \in P \cap Primitive : SelectedResource(T) \neq null$$

All resources in a schedule are unary, which means that they cannot execute more tasks simultaneously. Therefore,

in a feasible schedule for all selected primitive tasks $T_i \neq T_j$ the following holds:

$$SelectedResource(T_i) = SelectedResource(T_j) \Rightarrow E_i \leq S_j \vee E_j \leq S_i$$

Schedule

A schedule S (sometimes referred to as a resulting schedule or a solution) is acquired by determining the set P, and allocating the primitive tasks from P in time and on resources. Allocation in time means assigning particular values to the variables S_i and E_i for each $T_i \in P$. Allocation on resources means selecting a particular resource ($SelectedResource(T)$) from the resource group ($Resources(T)$) of each task $T \in P \cap Primitive$.

To make a schedule *feasible*, the allocation must be conducted in such a way that all the mentioned constraints in the problem are satisfied.

Rescheduling Problem

The problem we actually 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.

Formally, let $R = (Pr_0, Sch_0, \delta^+, \delta^-)$ be a rescheduling problem, which is given by the original scheduling problem Pr_0, the original feasible schedule Sch_0, elements δ^+ to be added to the problem Pr_0, and elements δ^- to be removed from the problem Pr_0. New scheduling problem Pr_1 is then $Pr_0 \cup \delta^+ \setminus \delta^-$. The task of the rescheduling problem R is then to find a schedule Sch_1 for problem Pr_1, the quality of which is measured with respect to the original schedule Sch_0.

The way the scheduling problem can be modified depends on the disturbance. In case of a resource failure, we are given a resource that cannot be used (from a certain time point) while the set of orders remains unchanged, therefore $\delta^+ = \emptyset$ and δ^- is the broken down resource.

Another example is an urgent order arrival, which is a disturbance where an order (a set of workflow instances) is added into the model, and the aim is to update the ongoing schedule in such a way that the added order is accomplished as early as possible. In this case δ^+ is the new order (including constraints among new tasks) and $\delta^- = \emptyset$.

As explained in the introduction, regardless of what the optimization objective of the original schedule is, it seems to be wise to modify the schedule in such a way that the new schedule is as similar to the original one as possible. For this purpose we need to evaluate the modification distance.

Let P_z denote the selected process P in the schedule Sch_z, and S_i^z denote the start time of task T_i in schedule Sch_z. Then, apart from computation time, we take into account the following distance functions:

$$f_0 = |\{T \in P_1 \setminus P_0\}| + |\{T \in P_0 \setminus P_1\}|$$

$$f_1 = \sum_{T_i \in P_0 \cap P_1 \cap Primitive} |S_i^1 - S_i^0|$$

$$f_2 = |\{T_i \in P_0 \cap P_1 \cap Primitive \mid S_i^1 \neq S_i^0\}|$$

$$f_3 = \max_{T_i \in P_0 \cap P_1 \cap Primitive} |S_i^1 - S_i^0|$$

In words, f_0 is the number of different tasks in the schedules, f_1 measures the total sum of time shifts of activities, f_2 counts the number of shifted activities, and f_3 is the biggest time shift of an activity. There are other conceivable distance functions, but we concentrate on these ones.

Current Work

Our recent work (Barták and Vlk 2015) proposes two methods to handle a resource failure occurring on the shop floor during the schedule execution. The first method, *Right Shift Affected*, 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, that is, minimizing the distance function f_2.

The second method, which is aimed at shifting activities by a short time distance regardless of the number of moved activities (that is, minimizing the distance functions f_1 and f_3), is called *STN-recovery*. The routine deallocates a subset of activities and then it allocates the activities again through integrating techniques from the field of constraint programming, namely *Conflict-Directed Backjumping with Backmarking* (Kondrak and Van Beek 1997). Before the allocation process, the search space is suitably pruned based on the values from the original schedule, which is another thing that seems to be neglected in the related literature.

The shortcoming of both the algorithms is that they neglect the possibility of alternative processes, which in practice may lead to a schedule recovery at a blow. Moreover, if the ongoing schedule is not recoverable, the algorithms are not able to securely report it and terminate.

Future Plans

We are currently developing algorithms under the hierarchical model described, where, in response to unexpected events, the intention is not only to modify the allocation of activities of the selected process, but to replace tasks in the process by other tasks that are not in the process, i.e., to re-plan some (ideally the smallest necessary, hence the motivation for the distance function f_0) subset of the schedule.

The first algorithm to try will work as follows. First, find the feasible process from all the orders in the schedule top-down, preferring the branches from the original schedule whenever possible. Second, after the process is selected, allocate activities from the process in time and on resources. If the second step fails, go back to the first step. Iterate until the schedule is found. For the second step, the crucial part of the STN-recovery algorithm mentioned above may be employed.

One natural improvement might be trying to allocate an activity straightaway when the corresponding primitive task is considered to be selected to the process. If the activity is successfully allocated, the searching for the process proceeds, otherwise it backtracks for alternatives.

However, our main effort will be made towards updating the schedule as locally as possible. If a resource suddenly becomes unavailable, it may suffice to merely replace the affected activities by alternatives that are just one level (one task decomposition) from the affected activities. It follows that the process should not be discarded and sought again top-down as described above, but it should be explored in a bottom-up fashion (from the primitive tasks upwards). The main difficulty of this approach are the logical constraints that must be propagated whenever the membership of a task in the process is being flipped.

In further future, the target is to extend the model of Nested Temporal Networks with Alternatives by recursion, and to suggest algorithms for this model. The recursion will bring the full power of planning, i.e., the possibility to generate tasks according to a given target. The main inspiration comes from the Hierarchical Task Networks (Nau et al. 2003). We will try modeling problems by attribute grammars, where modeling relations among attributes will be realized by constraint satisfaction problem rather than traditional semantic rules (Barták 2016).

Acknowledgments

This research is partially supported by SVV project number 260 224 and by the Czech Science Foundation under the project P103-15-19877S.

References

Abumaizar, R. J., and Svestka, J. A. 1997. Rescheduling job shops under random disruptions. *International Journal of Production Research* 35(7):2065–2082.

Barták, R., and Čepek, O. 2007. Nested temporal networks with alternatives. In *AAAI Workshop on Spatial and Temporal Reasoning, Technical Report WS-07-12, AAAI Press*, 1–8.

Barták, R., and Vlk, M. 2015. Machine breakdown recovery in production scheduling with simple temporal constraints. In *Agents and Artificial Intelligence*. Springer. 185–206.

Barták, R.; Jaška, M.; Novák, L.; Rovenský, V.; Skalický, T.; Cully, M.; Sheahan, C.; and Thanh-Tung, D. 2012. Flowopt: Bridging the gap between optimization technology and manufacturing planners. In *Luc De Raedt et al. (Eds.): Proceedings of 20th European Conference on Artificial Intelligence (ECAI 2012)*, 1003–1004. IOS Press.

Barták, R.; Muller, T.; and Rudová, H. 2003. Minimal perturbation problem-a formal view. *Neural Network World* 13(5):501–512.

Barták, R. 2016. Using attribute grammars to model nested workflows with extra constraints. In *SOFSEM 2016: Theory and Practice of Computer Science*. Springer. 171–182.

Cunningham, P., and Smyth, B. 1997. Case-based reasoning in scheduling: reusing solution components. *International Journal of Production Research* 35(11):2947–2962.

Dulai, T., and Werner-Stark, Á. 2015. A database-oriented workflow scheduler with historical data and resource substitution possibilities. In *Proceedings of the International Conference on Operations Research and Enterprise Systems*, 325–330. SciTePress.

Jain, A. S., and Meeran, S. 1998. Job-shop scheduling using neural networks. *International Journal of Production Research* 36(5):1249–1272.

Kondrak, G., and Van Beek, P. 1997. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence* 89(1):365–387.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. Shop2: An htn planning system. *J. Artif. Intell. Res.(JAIR)* 20:379–404.

Oddi, A.; Policella, N.; Cesta, A.; and Smith, S. F. 2007. Boosting the performance of iterative flattening search. In *AI* IA 2007: Artificial Intelligence and Human-Oriented Computing, LNCS 4733*. Springer Verlag. 447–458.

Ouelhadj, D., and Petrovic, S. 2009. A survey of dynamic scheduling in manufacturing systems. *Journal of Scheduling* 12(4):417–431.

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

Ramkumar, R.; Tamilarasi, A.; and Devi, T. 2011. Multi criteria job shop schedule using fuzzy logic control for multiple machines multiple jobs. *International Journal of Computer Theory and Engineering* 3(2):282–286.

Skalický, T. 2011. Interactive scheduling and visualisation. Master's thesis, Charles University in Prague.

Smith, S. F. 1995. Reactive scheduling systems. In *D. Brown and W. Scherer (eds.), Intelligent scheduling systems*, 155–192. Springer US.

Vieira, G.; Herrmann, J.; and Lin, E. 2003. Rescheduling manufacturing systems: a framework of strategies, policies, and methods. *Journal of Scheduling* 6:39–62.

Zhang, L.; Wong, T.; Zhang, S.; and Wan, S. 2011. A multi-agent system architecture for integrated process planning and scheduling with meta-heuristics. In *Proceedings of the 41st International Conference on Computers & Industrial Engineering*.