Budapest University of Technology and Economics
Department of Measurement and Information Systems

Hungarian Academy of Sciences
Computer and Automation Research Institute

Novel Models and Algorithms for Integrated Production Planning and Scheduling

Ph.D. Thesis

András Kovács

Supervisors:

József Váncza, Ph.D.
Computer and Automation Research Institute
Hungarian Academy of Sciences

Tadeusz P. Dobrowiecki, Ph.D.
Department of Measurement and Information Systems
Budapest University of Technology and Economics

Budapest, 2005.

Nyilatkozat

Alulírott Kovács András kijelentem, hogy ezt a doktori értekezést magam készítettem, és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2005. június 6.

.....
Kovács András

A dolgozat szövege stilisztikai megfontolásból nagyrészt első szám harmadik személyben íródott. A szerző saját eredményei a mellékelt tézisfüzet segítségével egyértelműen azonosíthatók.

A dolgozat bírálatai és a védésről készült jegyzőkönyv a későbbiekben a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karának Dékáni Hivatalában elérhetők.

Kivonat

Ebben a disszertációban a termeléstervezés és -ütemezés feladatával foglalkozunk a megrendelésre történő gyártás területén. Hatékony modellezési és megoldási technikákat keresünk, amelyek elősegítik egy vállalat termelékenységének és kiszolgálási szintjének növelését, a gyártási költségek csökkentését. Ehhez olyan módszerekre van szükség, amelyek képesek megfelelő döntéstámogatást nyújatni a menedzsmentnek a tervezési hierarchia e két szintjén.

A disszertációban az aggregált termeléstervezési feladat egy új modelljét definiáljuk, azzal a céllal, hogy olyan termelési terveket tudjunk előállítani, amelyeket megvalósítható részletes ütemtervvé lehet kifejteni. Ezt egy automatikus aggregációs eljárás segítségével érjük el, amely részletes termelési adatokból építi fel az aggregált reprezentációt, polinom idejű fapartícionálási algoritmusok segítségével.

Áttekintjük a részletes termelésütemezési feladatok megoldási lehetőségeit korlátozás-alapú ütemezés alkalmazásával. Kimutatjuk, hogy bár a korlátozás programozás a modellezési eszközök gazdag tárházát nyújtja az ütemezési feladatok leírásához, a feladatok optimális megoldásának megtalálása gyakran meghaladja a ma ismert algoritmusok képességeit. Ezért kutatási célként ezen algoritmusok hatékonyságának növelését tűztük ki, az iparban felmerülő ütemezési feladatok néhány tipikus strukturális tulajdonságának kihasználásával. E célból új, úgynevezett konzisztencia megőrző transzformációkat definiálunk.

Mindkét szinten hangsúlyt helyeztünk arra, hogy valós, gyakorlati relevanciával bíró feladatokat oldjunk meg. Kifejlesztettünk egy kísérleti integrált termeléstervező és ütemező rendszert, amelynek segítségével algoritmusainkat valós, ipari partnerünktől származó tervezési és ütemezési feladatokon tesztelhettük.

Abstract

This thesis is concerned with production planning and scheduling in make-to-order manufacturing system. We seek effective modelling and efficient solution techniques that can help increase the productivity and the service level of an enterprise, together with reducing production costs, by supporting the management to make smarter decisions on these two levels of the planning hierarchy.

In the thesis, we define a novel formulation of the aggregate production planning problem, with the objective of finding production plans that can be refined into feasible detailed schedules. We achieve this by constructing the aggregate representation from detailed production data in an automated way, by an aggregation procedure based on fast, polynomial-time tree partitioning algorithms.

We review the possibilities of solving detailed production scheduling problems by using constraint-based techniques. We point out that although constraint programming provides a rich collection of modelling tools for the description of scheduling problems, the solution of such problems often challenges the currently known algorithms. Hence, we aim at boosting the efficiency of these algorithms by the exploitation of structural properties commonly present in industrial problem instances. For this purpose, we define new, so-called consistency preserving transformations.

On both levels, we laid emphasis on solving real problems that arise in the industry. We developed a pilot integrated production planner and scheduler software, and used this system to test our algorithms on real-life planning and scheduling problems, originating from an industrial partner.

Contents

Chapter 1

Introduction

Advanced representation and solution techniques in production planning and scheduling received significant attention during the past decades, both from the part of research communities and the industry. This interest comes quite natural, regarding that these methods hold out a promise of increased productivity, better service level, higher flexibility, together with lower production costs. It is presumed that the above objectives can be reached by supporting the management to make smarter decisions on various levels of the planning hierarchy.

Despite the attractive prospects, only a few of the recent research results has migrated into everyday practice. Although advances in operations research and artificial intelligence led to the development of novel modelling and solution techniques, industrial applications often require more: on the part of the researchers, richer models and more efficient algorithms. This thesis is concerned with such issues.

1.1 The Planning Hierarchy in Make-to-order Systems

The planning functions in make-to-order manufacturing environments are generally described by the three-level hierarchy presented in Fig. 1.1. The levels of decision making are called *strategic* (or long-term), *tactical* (medium-term), and *operational* (short-term). Every member of the hierarchy is responsible for realizing the objectives that characterize the given level, and the decisions made at a certain stage become constraints for the lower levels [25, 33, 96].

Accordingly, planning on the strategic level concerns long-term decisions that determine the market competitiveness policy. Departing from the choice of plant locations and capacities, these decisions include make-or-buy choices, supply network planning, and capacity/facility planning. Based on demand forecasts and other mar-

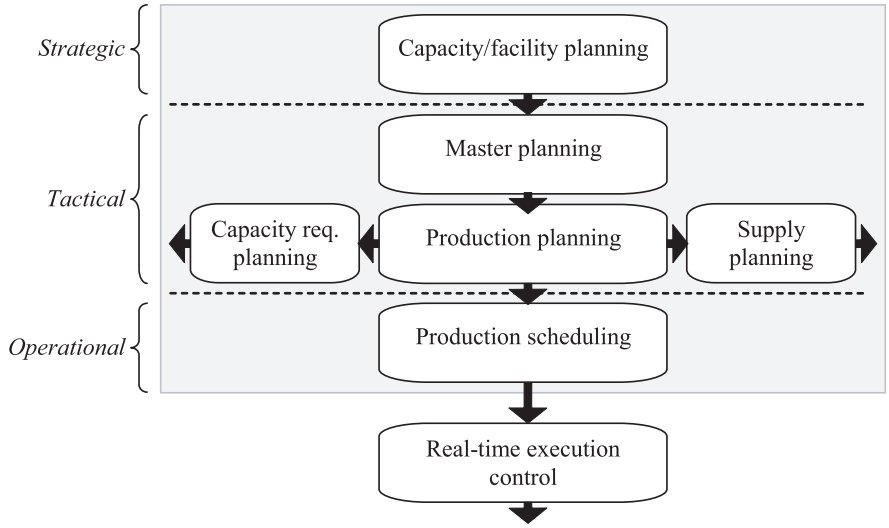


Figure 1.1: Levels of the planning hierarchy.

ket information, the required capacities of the machine resources, workforce, transportation means, etc. in the factory are also determined on this level. The decisions are brought by the senior management, over a planning horizon covering several years.

In contrast, the planning tasks of the tactical level are already directly related to customer orders, let them be contractual or only forecasted. The master planning module is responsible for the acceptance (or possibly, the rejection) of the orders, as well as for setting their due dates. Then, *production planning* assigns the production activities to time on an aggregate timescale. This assignment serves as the basis of the medium-term *material plan* that defines what and when raw materials should be purchased from the suppliers, and the *capacity plan* that determines the required amount of capacities per resources and aggregate time units. The capacity plan is of interest because the production capacities set on the strategic level can be temporarily increased by overtime, hired workforce, or subcontracting, or they can be decreased by granting leave to the workers. Depending on the industrial sector, the horizon of tactical planning ranges from 1 month to 1 year.

Finally, *production scheduling* on the operational level unfolds the first segments of the production plan into detailed resource assignments and operation sequences. Scheduling is performed on a detailed problem representation, for individual operations, with respect to fixed capacities. Under the production scheduling module, the presence of real-time execution control is often required. This can take place in the

form of a *Production Activity Controller*, or a *Manufacturing Execution System* that provides feedback about shop-floor status.

In this thesis, we focus on production planning and scheduling in make-to-order manufacturing systems. We assume that the exact description of the planning problem – the order set, resource capacities, raw material availability, and the detailed technological plans of the products – is known for the medium-term horizon. Note that this assumption excludes *engineer-to-order* companies from our scope, since they often prepare technological plans after order acceptance and production planning. We also assert that the presence of uncertainties is restricted enough to apply deterministic approaches. These assumptions will allow us to model the planning and scheduling problems as *combinatorial optimization problems* [78].

1.2 Problem Statement

Today, most factories apply *material requirements/manufacturing resources planning* (MRP) systems [96] for medium-term production planning. These systems focus on the material flow aspect of production, and assume that products can be manufactured with fixed lead times. Hence, they completely disregard the actual load on production capacities. No wonder that in an age characterized by market fluctuations, the plans generated this way can be barely unfolded to executable detailed schedules. Recently, several approaches have been suggested to couple the capacity and material flow oriented aspects of production planning [46, 56, 69]. A common characteristic of these models is that they apply a high-level description of the production activities and their complex interdependencies, which – in practice – has to be encoded manually, by a human expert. The high-level formalism does not always reflect the context of the underlying processes, and it cannot guarantee the feasibility of the production plans. Furthermore, the results depend largely on the proficiency and the mindfulness of the human modeler.

Our objective was to find a novel, aggregate formulation of the production planning problem which ensures that the generated plans can be refined into feasible detailed schedules. The representation of the planning problem should be generated automatically, from data readily available in de facto standard production databases.

The current industrial practice in production scheduling is also dominated by heuristic approaches, such as priority rule-based schedulers [57, 77]. In spite of this, well-known formal methods are available to describe what makes a schedule feasible,

and also to optimize the schedule according to various criteria. The most promising branch of these methods, *constraint-based scheduling* emerged in the early eighties [10, 37]. It offers a rich and straightforward representation to model even the finest details of the scheduling problem. However, the solution of the vast instances of the NP-complete combinatorial optimization problems that often arise in practice challenges currently known algorithms [97].

For short-term detailed scheduling, we decided for the application of the constraint-based approach. The objective of our research was to improve the efficiency of the currently known solution techniques, by the exploitation of typical structural properties of industrial problem instances. For this purpose, we applied so-called *consistency preserving transformations*.

During this research, we laid emphasis on solving real problems that arise in the industry. We developed a pilot integrated production planner and scheduler, and used this system to test our algorithms on real-life planning and scheduling problems, originating from an industrial partner.

1.3 Outline of the Thesis

The contents of this thesis are organized into four further chapters. In Chapter 2, we address modelling medium-term production planning problems in make-to-order project-oriented manufacturing systems. We introduce a novel, aggregate formulation of the production planning problem. Some preferable properties of the proposed representation are proven in the formal way, but a detailed analysis of its performance is presented through experiments on real-life production data.

Chapter 3 introduces constraint-based scheduling techniques for the solution of detailed production scheduling problems. We propose two novel methods – so-called consistency preserving transformations – to boost search on structured, practical problems. The efficiency of these transformations is illustrated by experimental results on industrial problem instances.

Chapter 4 is devoted to the demonstration of the industrial applicability of the proposed novel modelling and solution techniques. It presents a pilot production planner and scheduler system, named Proterv-II. The system is composed of a medium-term production planner and a short-term scheduler that apply the models and algorithms described in the first two chapters. The resistance of the schedules prepared by

deterministic techniques against various types of uncertainty was verified by discrete-event simulation.

A summary of the new results is presented and some further implications are pointed out in Chapter 5.

Finally, we highlight a naming convention that will be used throughout the thesis. While the words *operation*, *task*, and *activity* are often used as interchangeable in the scheduling literature, we make a clear distinction. By *operation* we mean the physical process to be performed in the factory. A *task* is the representative of an operation in the theoretical model of production scheduling. In contrast, *activities* are used in the medium-term production planning model, to denote a larger unit of work, usually built of several tasks.

Chapter 2

Aggregate Modelling of Production Planning Problems

The medium-term production planning level of the PPS hierarchy plays a fundamental role in determining both the service level and the production costs in make-to-order manufacturing systems. These manufacturing systems may execute hundreds of thousands of manufacturing operations under various capacity and technological constraints within the planning horizon. Hence, finding an executable production plan that meets project deadlines and keeps production costs low challenges any branch of operations research or artificial intelligence.

In current industrial practice, production planning is still based on the fixed lead time assumption of *material requirements/manufacturing resources planning* (MRP/MRP II) systems [96]. This assumption entails that neither resource capacities, nor raw material availability can be directly considered, and project lead times are set on the basis of historical data and other estimates. No wonder that in an age characterized by market fluctuations and ever shorter product life cycles, plans generated this way can barely be refined to executable schedules.

We believe that the key to the development of an advanced PPS system is finding the appropriate representation of the planning problem that captures both the material-flow and resource-oriented aspects of production. Clearly, planning on the medium-term horizon requires *aggregation*, i.e., merging the fine details of the production processes, in order to keep the computational complexity of the problem in a tractable range. Aggregation can be performed with respect to time, resources, and production activities.

However, an aggregate representation of the planning problem is legitimate only if it facilitates finding plans that can be unfolded into feasible detailed schedules.

This motivated our research to reveal the impact of modelling decisions made during the preparation of the aggregate representation on the quality of the final output of the PPS hierarchy: the executed detailed schedules. Results of these considerations led us to a novel, aggregate formulation of the production planning problem. The conception was validated by experiments on real-life production data originating from an industrial partner. We note that elements and precursors of the approach were originally published in [58, 59, 61, 95].

The chapter is organized as follows. First, we give an introduction to aggregate production planning and identify our objectives in Sect. 2.1. Then, in Sect. 2.2 we briefly present the applied mathematical models of the production scheduling and the production planning problems. In Sect. 2.3, we define our aggregate model of the production planning problem, and introduce an aggregation/disaggregation procedure to construct it from detailed production data. Polynomial-time tree partitioning algorithms are proposed for the creation of such aggregate models in Sect. 2.4. We discuss some subsidiary points and give an outlook on possible extensions in Sect. 2.5. Finally, we present the experimental results achieved on real-life problems in Sect. 2.6, and draw the conclusions in Sect. 2.7.

2.1 Introduction to Aggregate Production Planning

Aggregation is a widely used technique for reducing the computational complexity of combinatorial optimization problems [83]. Aggregate problem solving consists of three major steps. First, the *detailed model* of the problem is *aggregated*, i.e., several variables of the detailed model are replaced by one aggregate variable, and several constraints by one aggregate constraint. Then, the resulting *aggregate model* is solved by appropriate algorithms. Finally, in the *disaggregation* step, the results received on the aggregate level are projected back to the detailed level, see Fig. 2.1.

While sometimes the disaggregation step is trivial, in other cases it requires the explicit solution of the detailed problem in the presence of constraints derived from the aggregate solution. Note that the aggregate production planning level of our PPS architecture decomposes the medium-term problem into a sequence of disjoint weekly detailed scheduling problems that will be solved independently of each other, as presented in Fig. 2.2. In practice, detailed schedules will be generated for the next few weeks only.

An aggregation/disaggregation procedure is called *feasible*, if it ensures that any

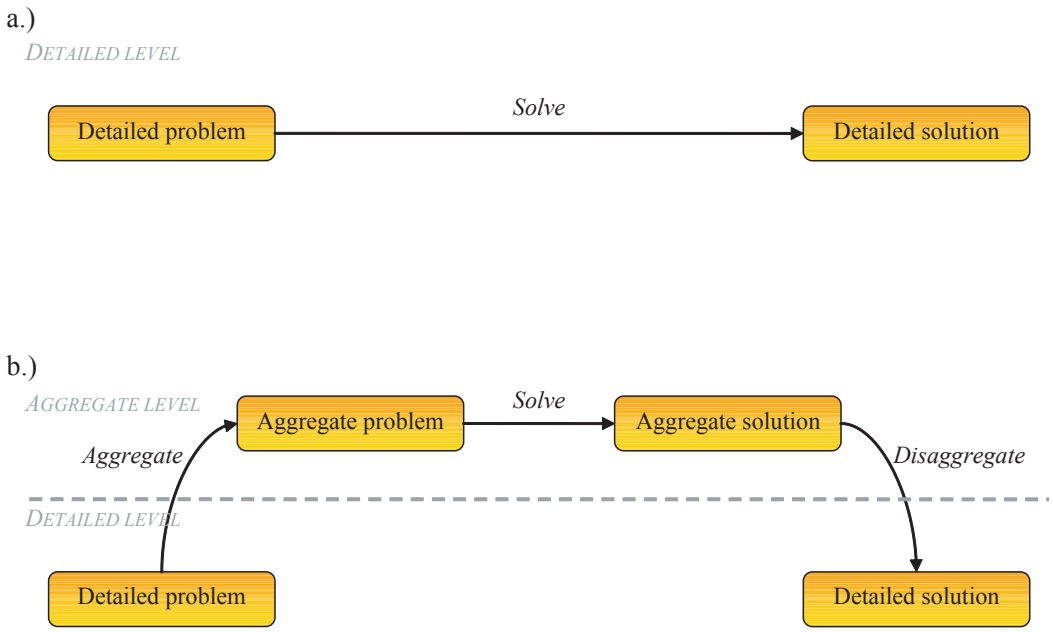


Figure 2.1: Solving a problem without (a.) and with (b.) aggregation.

feasible solution of the aggregate model can be disaggregated into a feasible solution of the original model. Furthermore, the procedure is called *optimal*, if the optimal solution of the aggregate model can be disaggregated into an optimal solution of the original model. Generally, aggregation involves a certain relaxation of the original problem, but often additional constraints are introduced, too. Consequently, feasibility and optimality of the aggregation/disaggregation procedure can rarely be guaranteed, and the quality of the approximation it provides constitutes a crucial issue.

Aggregation methodology has been extensively studied in the field of *linear programming* (LP), see [83] for a comprehensive overview. Well defined methods are available for the selection of the variables to merge – typically those that are in some respect similar –, as well as bounds on the loss of accuracy due to aggregation. Still, much less is known about aggregation in more expressive mathematical formulations, such as *mixed-integer linear programs* (MILP), see, for example, [44].

Specifically, in production planning, the idea of aggregation was introduced fifty years ago by Holt et al. [49], just with the motivation to respond to fluctuations in product orders by means of a clear-cut mathematical model using a common measure of work required by the individual orders. The classical approach for aggregate pro-

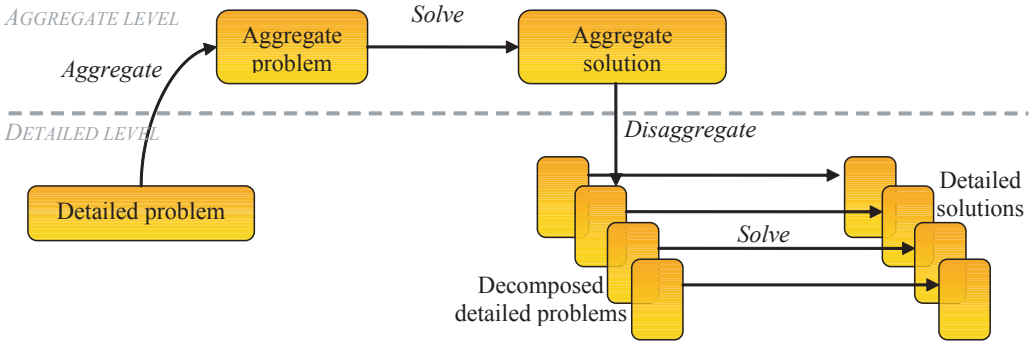


Figure 2.2: Our aggregate planning framework decomposes the detailed scheduling problem to weekly sub-problems.

duction planning in batch-type production systems was defined by Bitran et al. [16]. On the aggregate level of this hierarchical approach, production plans were prepared for families of similar products, instead of a large number of individual products. However, the linear programming formulation of the aggregate problem disregarded any temporal relationships between the various production activities, and the families were defined *a priori*.

The necessary and sufficient conditions of feasible aggregation in the latter model have been studied by Axsäter [3] and Erschler et al. [30]. However, these conditions reveal that feasibility (perfect aggregation, in the authors' terms) and optimality could be reached only under very specific circumstances. Toczyłowski and Pieńkosz [87] proposed a feasible aggregation procedure for the same production planning model. They achieved feasibility by assigning higher production cost, inventory holding cost, and resource requirements to product families than the appropriate cost of any product contained by the family. However, this approach could lead to serious sub-optimality in the case of significant variance within product families. A more recent paper by Leisten [71] reviews these aggregation/disaggregation procedures from the viewpoint of LP-aggregation. The author also investigates how feasibility and optimality can be approached by iteratively adjusting or refining the aggregate model.

For the case of project-oriented systems, a different approach that merges tasks requiring the same set of resources into one aggregate activity was suggested by Hackman and Leachman [43]. This simple way of aggregation can be easily understood by human experts, which makes it an ideal representation in, e.g., engineering-to-order manufacturing systems, where production planning has to be performed before the

preparation of the detailed technological plans, based on engineers' estimates.

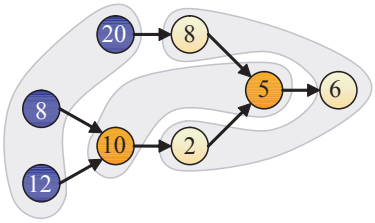


Figure 2.3: How to express the temporal interdependencies between these activities?

However, if parts loop over the same resources several times, this approach may result in very complex temporal interdependencies between the activities. Consider the example in Fig. 2.3, where vertices of the same color represent tasks that employ identical resources (blue vertices stand for components manufacturing, orange for assembly, and yellow for inspection). Durations of the tasks are indicated on the vertices, and the edges of the graph denote precedence constraints between the tasks. Even in this small example, we should be able to express that either assembly or inspection can start when 50% of components manufacturing is ready, but not both of them. The balanced progression of assembly and inspection is also a requirement. In [70], an extensive collection of constraints is suggested for the description of such temporal relationships. They use variable duration activities with prescribed intensity curves, overlap relationships, as well as balance-type relationships between the dependent activities. However, even this complex formulation cannot guarantee the feasibility of the aggregation/disaggregation procedure. Furthermore, this approach faces serious difficulties in obtaining the required input data that is seldom available in existing technological databases. Consequently, this modelling policy requires the involvement of human experts, even if their work can be supported by software systems exploiting the similarities between past and current projects [91].

In contrast to the above approaches, our goal was to define an aggregation/disaggregation framework for production planning in make-to-order project-oriented systems that fulfills the following requirements.

- Both aggregate planning and detailed scheduling must respect the main temporal constraints (e.g., project due dates and precedences) and the capacity constraints of the factory.

- Aggregation should reduce the complexity of the planning problem so that its close-to-optimal solution becomes possible in a reasonable amount of time.

- At the same time, the aggregation/diaggregation procedure should approach feasibility and optimality.

- The aggregate model of production planning must be generated from the same product and production related data that is used during detailed scheduling. Aggregation and disaggregation must be performed automatically, without the involvement of human experts.

2.2 Formal Models of Production

We address make-to-order manufacturing systems where the detailed production scheduling problem can be captured by the classical *resource-constraint project scheduling problem* (RCPSP) model [20]. In this representation, one fixed-duration task stands for each operation. The tasks compete for finite capacity resources, and it is assumed that the technological constraints among the operations of a project can be described solely by precedence relations between tasks. This model, to be presented hereinafter constitutes our detailed problem. Departing from this representation, aggregation is expected to generate a compact, aggregate production planning model.

In the medium-term planning problem, we consider project time windows strict, but we allow flexible capacities. Our optimization criteria are minimal extra capacity usage and minimal *work-in-process* (WIP). During the disaggregation step, the optimal solution of the aggregate problem is translated into a sequence of detailed scheduling problems, where the horizon of each problem corresponds to one aggregate time unit. In practice, only the schedules of the first few aggregate time units are of interest.

When solving the short-term scheduling problems, our objective is to achieve the goals set by the planner. Hence, we regard the resource capacities as fixed, and minimize makespan. Observe that the aggregation/disaggregation procedure is feasible for a given planning problem instance if and only if there exists a detailed solution for each induced short-term problem with a makespan not greater than the length of the aggregate time unit.

2.2.1 Production Scheduling: the RCPSP Model

In the detailed production scheduling problem, there is a set of *projects* $\mathbf{P}$ to be executed within the scheduling horizon. Each project $P \in \mathbf{P}$ is characterized by an *earliest start time* est_P and a *latest finish time* lft_P. The project P comprises a set of non-preemptive *tasks* T_P. The overall set of tasks is denoted by T, and each task $t \in T$ has a fixed *duration* d_t. Each task t requires one unit of the renewable cumulative *resource* $r(t) \in \mathbf{R}$ during the whole length of its execution. The *capacity* of the resource r is denoted by $q(r)$, which means that r is able to process at most $q(r)$ tasks at a time. Furthermore, tasks that belong to the same project can be connected by end-to-start *precedence constraints*. The precedence constraint $(t_1 \rightarrow t_2)$ states that task t_1 must end before the start of task t_2, i.e., $end_{t_1} \leq start_{t_2}$. Throughout this chapter we assume that the precedence constraints between the tasks of a project determine an in-tree together, which fits the needs of typical components manufacturing industries.

Then, the solution of an RCPSP instance consists of determining valid start times $start_t$ for the tasks such that all temporal, precedence, and resource constraints are satisfied and some objective function is minimized. Typical optimization criteria are minimizing the makespan, maximum tardiness, weighted tardiness, etc. The RCPSP problem with any of the previous optimization criteria is NP-complete in the strong sense. For an overview of the possible solution approaches, readers should refer to [17, 20].

When planning on the medium-term horizon, we consider strict project time windows and flexible capacities. The latter means that the normal capacity $q(r)$ of resource r can be extended by *extra capacities* – such as overtime or subcontracting –, at a cost proportional to the quantity and the duration of the usage. In our current settings, we minimize the cost of extra resource usage first, and WIP in the second run, with the previous bound on extra resource usage. Since in our specific application it was a basic assumption that all the raw materials of a project had to be on stock by the start time of the project, we applied the following formula to calculate WIP. In the formula, $start_P = \min_{t \in T_P} start_t$ stands for the start time of project P, and w_P is a project-specific weight factor:

$$\sum_{P \in \mathbf{P}} w_P \cdot (lft_P - start_P).$$

In the short-term scheduling problems we regard resource capacities as fixed and non-extendible. Capacities $q(r)$ are set to the values determined by the medium-term planner. We minimize the *makespan*, i.e., the maximum of the end times of the tasks. Finally, note that all parameters of this detailed scheduling model are directly available from de facto standard production databases.

2.2.2 Production Planning: RCPSP with Variable-intensity Activities

The proposed aggregate representation of the production planning problem is based on an extension of the resource-constrained project scheduling problem, suggested recently by Kis [56] and Márkus et al. [74], named *resource-constrained project scheduling problem with variable-intensity activities* (RCPSVP). It works with variable-intensity, fixed-volume activities and continuously divisible resources, which fits the needs of production planning better than the classical RCPSP, intended for detailed, job-shop level scheduling. We note that similar variable-intensity scheduling models have been discussed earlier by Hans [46] and Leachman et al. [69].

An instance of the RCPSVP problem is given by a finite set $\mathbf{P}$ of *projects*, a set $\mathbf{A}$ of *activities* that build up the projects, a set $\mathbf{R}$ of continuously divisible renewable *resources*, and a directed acyclic graph $\mathbf{G} = (\mathbf{A}; \mathbf{E})$ representing end-to-start *precedence constraints* between the activities. Each activity $A \in \mathbf{A}$ must be entirely processed between its *earliest start time* est_A and *latest finish time* lft_A. The time horizon is divided into discrete time units. In each time unit τ of the horizon, a portion x_τ^A of activity A is executed. We call x_τ^A the *intensity* of A in τ. Clearly, $\sum_\tau x_\tau^A = 1$ must hold. Furthermore, there is a *maximal intensity* j_A defined for each activity A.

Each activity may require the simultaneous use of some resources, proportionally to its intensity. Hence, if the entire processing of activity A requires a total work of ϱ_r^A on resource r, then it occupies $\varrho_r^A \cdot x_\tau^A$ units of this resource at time τ. Each resource $r \in \mathbf{R}$ has a *normal capacity* of q_τ^r units that is available free of charge, and it has an additional *extra capacity* of $\hat{q}_\tau^r$ units at the expense of c_τ^r for each extra unit used. The solution of the RCPSVP problem consists of determining an intensity x_τ^A for each activity A and time unit τ, such that the temporal and precedence constraints are fulfilled, the resource demand does not exceed the resource availability (normal + extra) in any time unit, and the total cost of extra capacity usage is minimized.

The RCPSVP problem is NP-complete in the strong sense, as it was proven by

Kis in [56]. The same paper proposes a mixed integer-linear program formulation and a branch-and-cut solution approach, using customized cutting planes. The algorithm is capable of solving the RCPSVP problem for optimization criteria which can be expressed as a linear function of the x_τ^A, or established with a dichotomic search. Beyond minimizing extra capacity usage, these criteria include minimizing makespan, maximum tardiness, weighted tardiness or work-in-process. As stated above, our primary optimization criterion is the minimal cost of extra capacity usage, while the secondary criterion is minimizing WIP.

In the following sections, we investigate how the parameters of this aggregate model can be computed from the detailed representation so that our aim, i.e., a feasible and nearly optimal aggregation is realized.

2.3 An Aggregate Formulation of the Production Planning Problem

Below we present our novel aggregation/disaggregation procedure for production planning for make-to-order project oriented manufacturing systems. We perform aggregation in the dimensions of activities and time. We assume that the time unit of aggregate planning, denoted by Θ is given a priori, while the aggregate model of production activities is to be computed. The objective, as stated earlier, is to approach the feasibility and optimality of aggregation as good as possible, together with keeping the computational complexity of the aggregate problem in a tractable range. We do not aggregate resources, because the computational complexity does not depend tightly on the number of resources. This determines only the number of constraints, but not of the variables in the MILP formulation. At the same time, the extension of the approach to resource aggregation is rather straightforward, this being the requirement in an application.

For the sake of simplicity, we assume that the detailed scheduling problem fits into the basic RCPSP model described in Sect. 2.2.1. We will propose refinements of the aggregation procedure covering various extensions of the detailed scheduling model later, in Sect. 2.5.3.

In our approach, aggregation is performed once, before solving the production planning problem. At this time, no exact temporal assignment of the production activities is known, except for the time windows of projects. Although we can suspect that subsequent tasks of a project follow each other without major time lags, we do

not know which tasks of other projects will be processed concurrently in the factory. For this reason, we aggregate production activities *for each project separately.*

2.3.1 The Aggregation/Disaggregation Procedure

In our detailed scheduling model, the precedence constraints between tasks belonging to the same project form an in-tree. Consequently, each project can be described by a rooted tree, the so-called *project tree*. The vertices of the project tree represent tasks of the project, while edges correspond to precedence relations between the tasks. Vertices with several sons stand for assembly operations, while those with a single son denote either machining operations or joining a purchased part to the workpiece. The execution of the project over time advances from the leaves towards the root that stands for the finishing operation of the end product. Fig. 2.4 shows a sample part whose project tree, together with the related technological data is presented in Fig. 2.5. We note that project trees in practice are often much larger. In our particular application they contained up to 500 vertices.

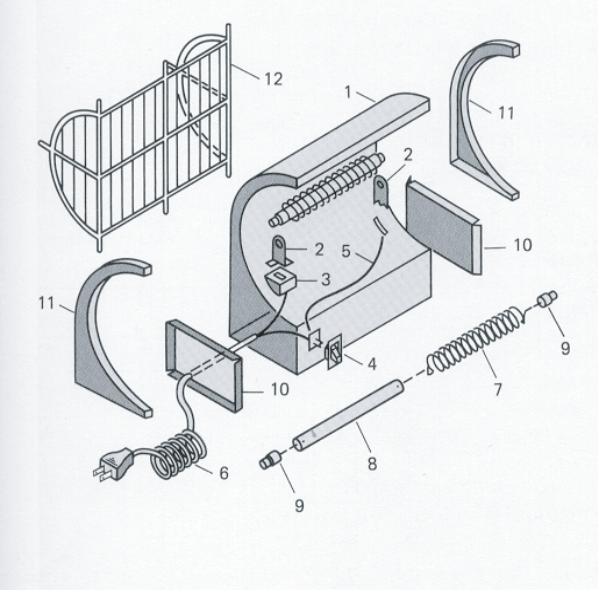
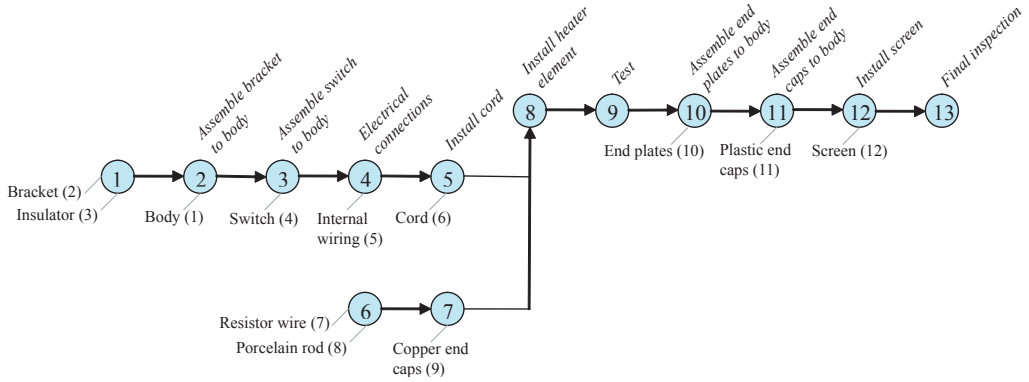


Figure 2.4: A sample part, adapted from [76, p. 179]

The aggregation procedure is based on *partitioning* the project trees into connected *sub-trees*, and merging the tasks that belong to the same sub-tree into one aggregate activity. Throughout the next pages we will give detailed considerations to the question of selecting the best suited partitioning. Nevertheless, once the par-



	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}
d_t	2.0	1.5	1.0	3.0	1.0	3.5	5.0	1.5	2.0	1.5	0.8	1.0	2.0
$r(t)$	r_1	r_1	r_1	r_1	r_1	r_2	r_2	r_3	r_1	r_1	r_1	r_1	r_3

Figure 2.5: Project tree of the sample part.

titioning of the project tree is determined for all the projects, the parameters of the aggregate problem can easily be computed as follows.

- For each precedence constraint $t_1 \rightarrow t_2$ in the detailed model, if tasks t_1 and t_2 are inserted into two *different* activities A_1 and A_2, then a precedence constraint $A_1 \rightarrow A_2$ is posted between the activities. Otherwise, the precedence constraint is omitted from the aggregate model. Observe that the graph of precedences among the activities will also form a tree.

- Let us denote the *minimum throughput time* of activity A by $d(A)$, and let μ_A be an *activity security factor*, to be discussed in detail in Sect. 2.3.2. Then, the maximal intensity of this activity is calculated as $j_A = \min(1, \frac{\mu_A \Theta}{d(A)})$.

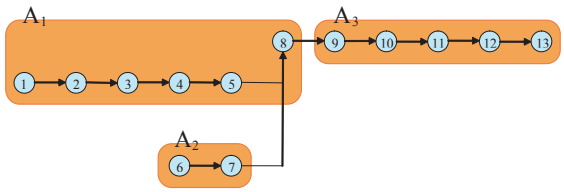
- The earliest start times est_A and latest finish times lft_A of the activities are set to the earliest start and latest finish times of the corresponding projects. We assume that these are integer multiples of the aggregate time unit length Θ. During the solution of the planning problem, the solver is able to deduce tighter time windows for the activities which are connected to others by precedence constraints.

- The aggregate model uses the same set of resources as the detailed representation. Aggregate resource capacities q^r_τ are computed as the integral of the

detailed capacities over the aggregate time unit τ, reduced by a *resource security factor* μ_R (see Sect. 2.3.2 for details). We suggest the application of infinite extra capacities in order to avoid unsolvable problem instances.

- Finally, the resource requirements of an activity are the sums of resource requirements of the contained tasks, i.e.,

$$\varrho_r^A = \sum_{t \in A:\ r(t)=r} d_t.$$



	$d(A)$	j_A	$\varrho_{r_1}^A$	$\varrho_{r_2}^A$	$\varrho_{r_3}^A$
A1	10.0	1.0	8.5	-	1.5
A2	8.5	1.0	-	8.5	-
A3	7.3	1.0	5.3	-	2.0

Figure 2.6: An aggregate model of the sample project.

A possible aggregate model of the sample project with an aggregate time unit length Θ of 10 is presented in Fig. 2.6. Now, solving the aggregate production planning problem consists of computing the intensities x_τ^A of such activities over time. For a given solution of the planning problem, the activities entirely processed within one aggregate time unit are called *complete*, while the activities whose execution is divided between several time units will be referred to as *broken*.

Disaggregation of the production plan involves ordering each task of the detailed model into one aggregate time unit. For the tasks of complete activities, the selection of time unit is unambiguous. In contrast, the tasks which belong to a broken activity are sorted by their decreasing distance from the root in the project tree, and are assigned to the time units designated for the activity proportionally to the intensity of the activity. The disaggregation of the production plan is complete with solving the detailed scheduling problems corresponding to the aggregate time units, for example by the techniques presented in detail in Chapter 3.

2.3.2 The Aggregate Model of Projects

Clearly, different partitionings of the project tree result in different aggregate representations of the production planning problem. In turn, different representations

bear different promises of feasibility and optimality. In this section we aim at identifying the characteristics that make an aggregate representation superior. Departing from this analysis, we arrive at the definition of the aggregate model of projects.

In a rather simplified view, the larger activities we apply in the aggregate model the more effectively we reduce the computational complexity of the planning problem – at the price of loosing the more of the accuracy of the representation. The most important respect of relaxation is disregarding the complex interactions among tasks of different activities ordered into the same aggregate time unit, and examining their resource requirements and activity throughput times separately.

The proposed aggregation procedure ensures that – if sufficient resources are available – processing the tasks of an activity fits into the designated aggregate time units. Nevertheless, the strongest formal statement that can be made about the satisfaction of resource constraints in the detailed solution is that the total load on each resource in each period corresponding to an aggregate time unit does not exceed the total available capacity of the resource. This statement holds only if there are no broken activities in the aggregate solution, since the resource requirements of tasks contained by broken activities are partially considered in aggregate time units other than the one in which they will be executed. For this reason, we set the upper bound of activity throughput times to Θ, the length of the aggregate time unit, and apply maximal intensities of $j_A = 1$. These settings lead to aggregate plans where only a negligible portion of the activities are broken.[1] Furthermore, in our specific application, this choice of activity size resulted in aggregate problems with a manageable computational complexity. Tasks whose duration was larger than Θ were ordered into a single activity.

Clearly, all the above considerations are necessary, but not sufficient conditions of the feasibility of the aggregation procedure. To help this, we introduce activity and resource security factors, denoted by μ_A and μ_R, respectively. Then, the upper bound on activity throughput times is set to $\mu_A \cdot \Theta$, where $\mu_A \leq 1$, while resource capacities are scaled down by a factor of $\mu_R \leq 1$. Note that the two security factors

[1]The explanation of the low ratio of broken activities lies in the MILP problem formulation, where the real variables are the intensities x_τ^A, and the inequalities defined on them are $x_\tau^A \geq 0$, $\sum_\tau x_\tau^A = 1$, and the resource constraint $\sum_A \varrho_r^A \cdot x_\tau^A \leq q_\tau^r$.

For these real variables, the simplex-based solver returns a basic solution [93], which, in our case means that an activity can be broken only if it employs a fully loaded resource in at least one time unit. In addition, the optimization criterion of minimal WIP also entails executing the competing activities sequentially, rather than in parallel. In experiments, all these effects resulted in a ratio of broken activities of between 2% and 7.5%, even for strongly resource-constrained problem instances.

differ essentially. Roughly speaking, μ_A corresponds to the expected portion of gaps in the *project view* of the Gantt chart representation of the detailed solution, μ_R is related to the gaps in the *resource view*. Hence, μ_A equals 1 and μ_R is low when unlimited resources are available for processing few tasks connected by precedence constraints, and vice versa for the case where many tasks are to be executed on scarce resources.

Nevertheless, the aggregate model is not a clear-cut relaxation of the detailed representation. During the aggregation step, new constraints are introduced as well, which leads to loosing the *optimality* of the aggregation. These new constraints derive from the aggregation of time in the discrete-time representation: a precedence constraint states that the two corresponding activities have to be executed in the given order, *in distinct time units*. Therefore, the throughput time of a project using a given partitioning is at least one greater than the *height of the partitioning*, i.e., the number of edges on the longest directed path of precedences in the aggregate project model. Consider the alternative partitionings of the sample project tree in Fig. 2.7. While the first necessitates a time window of at least 3 time units, the second enables us to execute the project with a throughput time of 2. Obviously, in order to respect the time windows of the projects and to keep WIP low, we are interested in finding aggregate project models with minimal height. In another point of view, this means increasing the parallelism between activities. After all the above considerations, we are ready to give the definition of the aggregate model of a project.

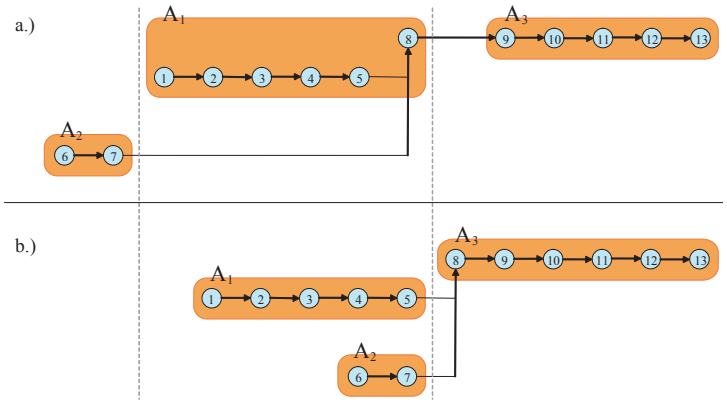


Figure 2.7: Executions of two different aggregate models of the sample project over time. Model (a.) requires at least 3 time units, while the minimal-height aggregate model (b.) needs 2 time units only.

Definition 2.1 *The aggregate model of a project is a partitioning of the project tree into connected sub-trees such that*

- *the throughput times of the activities corresponding to the sub-trees respect the upper bound of* $\mu_A \cdot \Theta$*, which helps us approach feasibility of the aggregation;*
- *the height of the partitioning is minimal, in order to ensure the optimality of the aggregation w.r.t. the criterion of minimal WIP;*
- *with the above prerequisites, the cardinality of the partitioning is also minimal, so that the aggregate model is kept as compact as possible.*

2.3.3 Feasibility and Optimality of the Aggregation

Up to now, we have defined the aggregate model of projects, and asserted that even these models cannot formally guarantee the feasibility or optimality of the aggregation procedure. Below we characterize qualitatively the approximation of feasibility and optimality that can be reached by the proposed framework. The experimental results achieved on real-life problem instances will be presented later, in Sect. 2.6.

Throughout the section we assume that all activity and resource security factors are fixed to 1. In practice, security factors lower than 1 facilitate finding a feasible detailed solution, but may worsen the objective value of such a solution.

Definition 2.2 *An aggregation/disaggregation procedure is* time-feasible *if and only if any aggregate solution has a disaggregation in which all temporal constraints, i.e., precedences and project time windows are respected.*

Theorem 2.1 *The aggregation procedure presented in Sect. 2.3.2 is time-feasible.*

Proof: Project time windows are observed by the construction of the aggregate problem. The satisfaction of precedence constraints between tasks belonging to distinct activities is ensured by precedence constraints among the corresponding activities. Furthermore, with resource constraints omitted, there exists a detailed schedule for each aggregate time unit where the precedence constraints are respected, because the throughput times of individual activities are at most Θ. □

Definition 2.3 *An aggregation/disaggregation procedure is defined* resource-feasible per aggregate time unit *if any aggregate solution can be disaggregated into a detailed*

schedule in which the overall demand for each resource is at most the total capacity of the resource in time intervals corresponding to aggregate time units.

Theorem 2.2 *The aggregation procedure presented in Sect. 2.3.2 is resource-feasible per aggregate time unit for solutions that do not contain broken activities.*

Proof: In the aggregate-level solutions, the sum of resource requirements does not exceed the available capacity in any aggregate time unit. If there are no broken activities in the aggregate solution, then the resource requirement of each task is completely considered in the aggregate time unit into which the task will be ordered during disaggregation. Hence, resource-feasibility per aggregate time unit will hold for the detailed solution, too. □

After the above statements on the approximation of feasibility, we address the optimality of the aggregation/disaggregation procedure according to the criterion of minimal WIP, still with resource constraints omitted. For this purpose, let us denote the optimal aggregate plan by Π^*, and the optimal detailed schedule by Γ^*. Note that, Π^* has a feasible disaggregation by Theorem 2.1, which will be denoted by Γ_{Π^*}.

Theorem 2.3 *If resource constraints are omitted, then it holds that* $WIP(\Gamma_{\Pi^*}) \leq WIP(\Gamma^*) + \Theta \cdot \sum_{P \in \mathbf{P}} w_P$.

Proof: With resource constraints omitted, both the optimal aggregate plan Π^* and the optimal detailed schedule Γ^* consist of activities/tasks *shifted right* towards the latest finish time of the project, as far as this is allowed by the precedence constraints within the project.

Observe that Γ^* induces a partitioning of each project tree, in which a component is the set of tasks of a project which are executed within one aggregate time unit. All these components respect the upper bound of Θ on the throughput time of the corresponding activity. However, the height of the partitionings induced by Γ^* cannot be lower than that of the minimal-height partitionings applied for the preparation of Π^*. This implies that the actual start times of projects in Γ^* fall into the segment of the detailed-level horizon that corresponds to the aggregate-level start time of the project in Π^*. Hence, in Γ_{Π^*}, each project starts at most Θ earlier than in Γ^*. Consequently, we have $WIP(\Gamma_{\Pi^*}) \leq WIP(\Gamma^*) + \Theta \cdot \sum_{P \in \mathbf{P}} w_P$. □

2.4 Tree Partitioning Algorithms for the Creation of Aggregate Project Models

In the previous sections we have arrived at the conclusion that optimal aggregate project models can be constructed by partitioning the project tree into connected components that correspond to the activities of the project. Tree partitioning, in the presence of various constraints and optimization criteria constitutes a widely studied class of problems. It has numerous applications, e.g., in telecommunication networks design, vehicle routing or database paging.

We consider tree partitioning problems where a tree has to be split into disjoint sub-trees (components) that respect a certain weight limit. Our optimization criteria are the *minimal height* and the *minimal cardinality* of the partitioning, as well as the *Pareto bi-criteria* composed of these two. We begin by briefly reviewing the related literature.

For the problem where the weight of a component is calculated as the sum of the weights of the contained vertices, Kundu and Misra gave a linear time algorithm to minimize the cardinality of the partitioning [66]. For the same weight function, algorithms for minimizing height in linear time and determining the set of Pareto optimal solutions according to the bi-criteria of minimal height and minimal cardinality in polynomial time were suggested by Kovács and Kis [61]. Herein, we unite these algorithms in a common bottom-up framework and generalize the results for a larger set of component weight functions.

A related approach is the *shifting algorithm* of Becker and Perl [15] that partitions trees into a fixed number of components in the face of a wide choice of optimization criteria and component weight functions. Embedded in a dichotomic search, this algorithm is suitable for solving the minimum cardinality problem, however, with a significantly higher time complexity. Maravelle et al. [73] investigate several optimization criteria involving dissimilarities within or between components. Generalizations in which there are multiple weight or utility functions defined on the components are analyzed and solved by Hamacher et al. [45] and Johnson and Niemi [53].

2.4.1 Notations and Terminology

In the sequel $T = (V, E, r)$ always denotes a rooted tree with *vertex-set* V, *edge-set* E, and *root* r. The *sons* of a vertex $v \in V$ will be denoted by $S(v)$, noting that $S(v) = \emptyset$ if and only if v is a leaf. Let $T(v)$ be the sub-tree of T rooted at v consisting of v

and all vertices down to the leaves. The following definitions apply to T and also to all $T(v)$. $P = \{ST_1, \ldots, ST_q\}$ is a *partitioning* of T if and only if

- each component ST_i is a sub-tree of T,
- the ST_i are disjoint, and
- the union of the vertex-sets $V(ST_i)$ of the ST_i equals V.

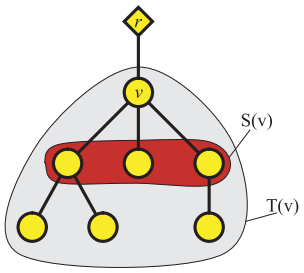


Figure 2.8: $S(v)$ denotes the sons of vertex v, $T(v)$ stands for the maximal sub-tree rooted at v.

The *cardinality* of a partitioning P of T is defined as $q(P) = |P|$. Each ST_i is rooted at the vertex closest to r in T. The *root component* of P is the one containing r, and will be denoted by $RC(P)$. For any partitioning P of T, let T^P denote the rooted tree obtained from T by contracting each $ST_i \in P$ into a vertex. The *height* $h(T)$ *of a rooted tree* T is the maximum number of edges of paths having one end at the root. The *height* $h(P)$ *of a partitioning* P is the height of T^P.

There is also given a component weight function $w : ST \longrightarrow \mathbb{R}_+$ on the sub-trees of T and a constant W. We say that a partitioning $P = \{ST_1, \ldots, ST_q\}$ of T is admissible if and only if $w(ST_i) \leq W$ for every $ST_i \in P$. We assume that $w(\{v\}) \leq W$ for each $v \in V$, which implies that trees always have admissible partitionings. Furthermore, we introduce the notation of $rw(P) = w(RC(P))$ for the weight of the root component of P. Finally, we define two properties to characterize component weight functions.

Definition 2.4 *We call a component weight function* w monotonous *if and only if for any two sub-trees* ST_1 *and* ST_2 *of* T *such that* $ST_1 \subseteq ST_2$, $w(ST_1) \leq w(ST_2)$ *holds.*

Now, let ST_1, ST_2, ST_1' and ST_2' denote sub-trees of T such that ST_1 and ST_1' are rooted at $v \in V$, ST_2 at $u \in S(v)$, while ST_2' at $u' \in S(v)$ ($u \equiv u'$ is allowed). Furthermore, suppose that $ST_1 \cap ST_2 = \emptyset$ and $ST_1' \cap ST_2' = \emptyset$, see Fig. 2.9.

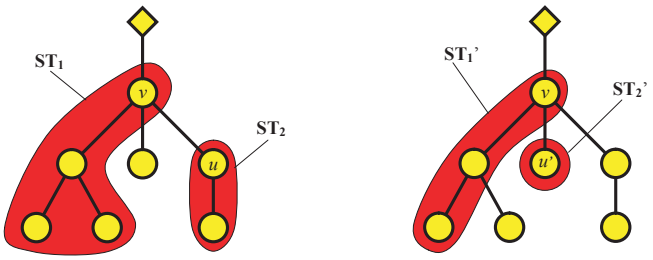


Figure 2.9: Illustration of the invariant property.

Definition 2.5 *A component weight function is called* invariant *if it is monotonous and for any sets of sub-trees that satisfy the above conditions, it holds that*

$$w(ST_1) \leq w(ST_1') \ \wedge w(ST_2) \leq w(ST_2') \ \Rightarrow \ w(ST_1 \cup ST_2) \leq w(ST_1' \cup ST_2').$$

2.4.2 The Bottom-up Framework

Our different algorithms follow a common bottom-up framework. In the initialization step, the algorithms assign the partitioning $P_v = \{\{v\}\}$ to each leaf v of T, which is the only partitioning of $T(v)$. In the iterative step, an arbitrary unprocessed vertex v is chosen, all of whose sons have already been processed. The (set of) optimal partitioning(s) of $T(v)$ are built by using optimal partitionings of the trees $T(u)$, $u \in S(v)$. This step is repeated until r is reached, at which point the (set of) optimal partitioning(s) of T is found.

During the iterative step, the partitioning P_v of $T(v)$ is obtained by applying the following *comb* operator to partitionings of the $T(u)$, $u \in S(v)$. Namely, let P_u be a partitioning of $T(u)$ and $K \subseteq S(v)$, then

$$P_v := comb(\{P_u \mid u \in S(v)\}, K)$$

is a partitioning of $T(v)$ that consists of all the components of all P_u except the root components of those P_u with $u \in K$, which together with v constitute the root component of P_v. See Fig. 2.10 for an illustration.

Observe that *any* partitioning P_v of $T(v)$ can be created by the *comb* operator applied on suitably selected partitionings P_u and set of sons K. Furthermore, the

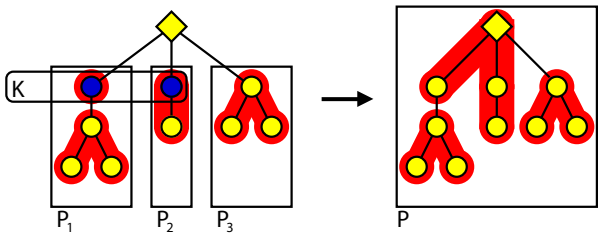


Figure 2.10: The *comb* operator applied to selected partitionings of the sons of the root.

$P(u)$, $\forall u \in S(v)$ and K is *unambiguously defined*. By consecutively applying this statement, we can deduce that $\forall u \in T(v)$ there exists *exactly one* partitioning P_u of $T(u)$ from which P_v can be built up by the iterative application of the *comb* operator. For a given vertex $u \in T(v)$, this partitioning P_u will be named the *u-generator* of P_v . The v-generator of P_v is itself. Now, the height and the cardinality of P_v can be calculated from the heights and the cardinalities of its generators as follows.

$$h(P_v) = \max\{\max_{u \in K} h(P_u), \max_{u \in S(v) \setminus K} h(P_u) + 1\} \quad (2.1)$$

$$q(P_v) = \sum_{u \in S(v)} q(P_u) - |K| + 1 \quad (2.2)$$

In the sequel, we describe some basic properties of partitionings and the *comb* operator.

Lemma 2.1 *If w is monotonous, then all the generators of an admissible partitioning are admissible.*

Proof: Suppose P is admissible, and P' is its generator. Then, for each sub-tree $ST' \in P'$ there exists a sub-tree $ST \in P$ such that $ST' \subseteq ST$. Hence, $w(ST') \leq w(ST) \leq W$ holds, which means that P' is admissible, too. □

Now, let us denote the minimal height and minimal cardinality of the admissible partitionings of $T(v)$ by $h_{min}(T(v))$ and $q_{min}(T(v))$, respectively.

Lemma 2.2 *If w is monotonous, $v \in V$ and $u \in T(v)$, then $h_{min}(T(v)) \geq h_{min}(T(u))$ and $q_{min}(T(v)) \geq q_{min}(T(u))$.*

Proof: Let P_v be a minimal height admissible partitioning of $T(v)$ and P_u its u-generator. Then, P_u is admissible (see Lemma 2.1) and its height is at most $h(P_v) = h_{min}(T(v))$, according to equation 2.1. For cardinalities, the proof is analogous. □

Finally, suppose that P_u and P'_u are admissible partitionings of $T(u)$ for each $u \in S(v)$, such that $\forall u: \ rw(P_u) \leq rw(P'_u)$ holds. Let $P_v := comb(\{P_u \mid u \in S(v)\}, K)$ and $P'_v := comb(\{P'_u \mid u \in S(v)\}, K)$, where K is an arbitrary subset of $S(v)$.

Lemma 2.3 *If w is invariant, then $rw(P_v) \leq rw(P'_v)$. Moreover, if P'_v is admissible, then so is P_v.*

Proof: Suppose that the lemma is false, and let K be a minimal subset of $S(v)$ (w.r.t. set inclusion) for which the required statement does not hold. Note that K is not empty, because then $rw(P_v) = rw(P'_v) = w(\{v\})$ would hold. Now, let u^* be an arbitrary vertex in K, and let

$$P^*_v := comb(\{P_u \mid u \in S(v)\}, K \setminus \{u^*\})$$

$$P'^*_v := comb(\{P'_u \mid u \in S(v)\}, K \setminus \{u^*\})$$

Then $rw(P_{u^*}) \leq rw(P'_{u^*})$ by definition, and $rw(P^*_v) \leq rw(P'^*_v)$ because K was a minimal set to contradict our statement. Since $RC(P_v) = RC(P_{u^*}) \cup RC(P_{v^*})$ and $RC(P'_v) = RC(P'_{u^*}) \cup RC(P'^*_v)$, it follows from the definition of the invariant property that $rw(P_v) \leq rw(P'_v)$, which is a contradiction.

Finally, P_v is admissible because it consists of components that are present in its admissible generators and a root component, for which $rw(P_v) \leq rw(P'_v) \leq W$ holds. □

In the following three sections, we describe three algorithms built on this bottom-up framework. The algorithms address three different optimization criteria. Table 2.1 gives an overview of the optimization criteria, the weight function properties that the different algorithms exploit, as well as the time complexity of the algorithms.

2.4.3 Minimizing the Height of the Partitioning

Below we describe how the parameters of the *comb* operator are chosen in the iterative step of the algorithm in order to minimize the height of the partitioning. Throughout this section, we assume that the applied component weight function w is *monotonous*.

Optimization criterion	Comp. weight function	Time complexity
Minimal height	Monotonous	$O(n)$
Minimal cardinality	Invariant	$O(n)$
Pareto min. height, min. card.	Invariant	$O(n^4)$

Table 2.1: Required properties of component weight functions and complexity of the algorithms.

We define the level $l_{P_v}(u)$ of a vertex $u \in T(v)$ with respect to a partitioning P_v of $T(v)$ as the height of the u-generator of P_v. Hence, if u_1 and u_2 are vertices within the same component of a partitioning P_v, then $l_{P_v}(u_1) = l_{P_v}(u_2)$. At the same time, if they are in distinct components, and u_2 is located on the path between u_1 and r, then $l_{P_v}(u_1) < l_{P_v}(u_2)$ holds. See Fig. 2.11 for illustration.

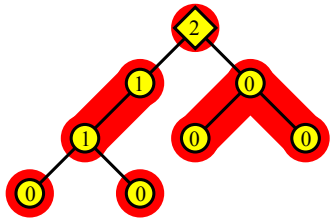


Figure 2.11: Levels of vertices w.r.t a partitioning.

Our bottom-up algorithm will assign a partitioning P_v^* of $T(v)$ to each vertex $v \in V$ in a way that $l_{P_v^*}(u)$ will be minimal for each $u \in T(v)$ over all admissible partitionings of $T(v)$. In other words, all the generators of the assigned partitioning will be minimal-height partitionings of the corresponding sub-trees $T(u)$. Hence, the partitioning assigned to r in the final step of the algorithm will be a minimal-height partitioning of T.

Note that the partitionings created in the initial step of the algorithm hold the above property. Then, in the iterative step, an unprocessed vertex $v \in V$ is chosen, for all of whose sons $u \in S(v)$ a partitioning P_u^* of $T(u)$ with the above property is known. Then let $h_{max} = \max_{u \in S(v)} h(P_u^*)$ and $K = \{u \mid u \in S(v) \wedge h(P_u^*) = h_{max}\}$. Furthermore, let

$$\begin{aligned} P_v^+ &= comb(\{P_u^* \mid u \in S(v)\}, K) \\ P_v^- &= comb(\{P_u^* \mid u \in S(v)\}, \emptyset) \end{aligned}$$

Note that $h(P_v^+) = h_{max}$ and $h(P_v^-) = h_{max} + 1$. Now, if P_v^+ is admissible, then the algorithm assigns $P_v^* = P_v^+$ to v, otherwise $P_v^* = P_v^-$. P_v^- is always admissible, because it consists of sub-trees that are present also in one of the admissible partitionings P_u^*, and $\{v\}$, for which $w(\{v\}) \leq W$ holds. Hence, our algorithm always assigns *admissible* partitionings to the vertices.

Lemma 2.4 *For each $u \in T(v)$ (including v), the u-generator of P_v^* is a minimal-height partitioning of $T(u)$.*

Proof: The proof of this statement for vertices apart from v is trivial, because the partitionings P_u^* for $u \in S(v)$ where selected to hold this property. In order to prove the statement for v as well, let us denote the set of v and all the vertices of components rooted at elements of K by L:

$$L := \{v\} \cup \{y \mid y \in T(u) \wedge u \in K\}$$

Now, if $w(L) \leq W$ then P_v^+ is admissible, and it is of minimal height, because no partitioning of $T(v)$ with smaller height exists according to Lemma 2.2. Otherwise, i.e., if $w(L) > W$ then the vertices of L cannot all belong to the same component. Note that for each $u \in L$ and for each admissible partitioning P_v of $T(v)$, $l_{P_v}(u) \geq h_{max}$ holds. Accordingly, there is a vertex $u \in L \backslash \{v\}$ with $l_{P_v}(u) \geq h_{max}$ in a separate component than the component containing v. Hence, the level of v is strictly larger than the level of this vertex u, i.e., $l_{P_v}(v) \geq h_{max} + 1$ holds in all the admissible partitionings P_v of $T(v)$. Consequently, $P_v^* = P_v^-$ is a minimal-height partitioning of $T(v)$, because its height is exactly $h_{max} + 1$. □

Finally note that the algorithm provides an admissible – though not necessary optimal – partitioning even if w is non-monotonous. The running time of the algorithm – assuming that w can be evaluated in unit time – is linear in the size of T.

2.4.4 Minimizing the Cardinality of the Partitioning

In this section we present how our bottom-up framework can be applied to minimizing the *cardinality* of the partitioning when the component weight function w is *invariant*. The algorithm will assign an admissible partitioning P_v^* of $T(v)$ to each vertex $v \in V$ whose cardinality is minimal over the admissible partitionings of $T(v)$, and whose root component weight $rw(P_v^*)$ is minimal among the minimal-cardinality admissible partitionings. A partitioning with this property will be briefly named *optimal*.

Suppose the optimal partitionings P_u^* of $T(u)$ are known for each vertex $u \in S(v)$, and the optimal partitioning P_v^* of $T(v)$ is to be constructed. Our algorithm sorts the sons u of v by increasing $rw(P_u^*)$ and assigns indices accordingly, i.e.,

$$rw(P_{u_1}^*) \leq rw(P_{u_2}^*) \leq \ ... \ \leq rw(P_{u_{|S(v)|}}^*)$$

will stand. Then, it computes subsets of $S(v)$, $K_i := \{u_j \mid 1 \leq j \leq i\}$ for $i = 1, ..., |(S(v)|$. Note that the ith subset contains i vertices $u \in S(v)$ with the smallest $rw(P_u^*)$. It selects the largest index i for which the root component of the partitioning $P_v^i := comb(\{P_u^* \mid u \in S(v)\}, K_i)$ respects the weight limit of W, and assigns this partitioning P_v^i to v, i.e., $P_v^* := P_v^i$.

The resulting partitioning P_v^* is *admissible*, because its generators are admissible and its root component was created to respect the weight limit of W. We prove its *optimality* in two steps. First, we show that the optimal partitionings P_u^* of the sub-trees $T(u)$ for $u \in S(v)$ can be used as the generators of an optimal partitioning of $T(v)$. Next, we prove that our algorithm combines these generators in a correct way, i.e., it selects parameter K of the *comb* operator appropriately.

Lemma 2.5 *For any admissible partitioning P_v of $T(v)$ there exists an admissible partitioning P_v' built from the optimal partitionings $P_u^*, u \in S(v)$ such that $q(P_v') \leq q(P_v)$ and $rw(P_v') \leq rw(P_v)$.*

Proof: Suppose $P_v = comb(\{P_u \mid u \in S(v)\}, K)$. Then let $P_v' := comb(\{P_u^* \mid u \in S(v)\}, K')$, where $K' := \{u \in K \mid q(P_u) = q(P_u^*)\}$. Since $\forall u \in S(v) : \ q(P_u^*) \leq q(P_u)$ and $\forall u \in K \setminus K' : \ q(P_u^*) \leq q(P_u) - 1$, we have

$$\textstyle\sum_{u \in S(v)} q(P_u^*) \leq \sum_{u \in S(v)} q(P_u) - |K \setminus K'|.$$

Now, the cardinality of P_v' can be calculated as follows (see equation 2.2).

$$\begin{array}{lcll} q(P_v') & = & \sum_{u \in S(v)} q(P_u^*) & -|K'| + 1 \\ q(P_v') & \leq & \sum_{u \in S(v)} q(P_u) - |K \setminus K'| & -|K'| + 1 \\ q(P_v') & \leq & \sum_{u \in S(v)} q(P_u) - |K| + 1 & \\ q(P_v') & \leq & q(P_v) & \end{array}$$

Furthermore, since w is invariant (and monotonous), $K' \subseteq K$ and $\forall u \in K'$: $rw(P_u^*) \leq rw(P_u)$ implies $rw(P_v') \leq rw(P_v)$, according to Lemma 2.3. Finally, P_v' is admissible, because it consists of components present in the admissible partitionings P_u^* and a root component, for which $rw(P_v') \leq rw(P_v) \leq W$ holds. □

Lemma 2.6 *The partitioning P_v^* computed by our algorithm is optimal among the partitionings of $T(v)$ that can be combined from the optimal partitionings P_u^* for $u \in S(v)$.*

Proof: Cardinalities of the partitionings of $T(v)$, which can be combined from the given partitionings P_u^*, depend only on the number of elements in the vertex set K (c.f. equation 2.2). For a given $|K|$, the cardinality of the partitionings is fixed. At the same time, the higher the $|K|$, the lower the cardinality of the partitioning.

It follows directly from Lemma 2.3 that for any fixed $|K|$, the root weight of the resulting partitioning is the lowest when $K = \{u_1, ..., u_{|K|}\}$. Our algorithm examines exactly this kind of vertex sets K. Furthermore, the algorithm selects the highest $|K|$ that is allowed by the component weight limit of W. □

Therefore, the partitioning P_v^* created by our algorithm is a minimal-cardinality (and minimal root weight) admissible partitioning of $T(v)$. The algorithm in its above form runs in $O(n \log n)$ time, since the sons u of each vertex have to be sorted by increasing $rw(P_u^*)$. However, parameter K of the *comb* operator can be computed without explicitly sorting $S(v)$, by using a method proposed in [66]. This method is based on the successive application of a linear-time median finding algorithm [18]. This way, the time complexity of the overall algorithm reduces to $O(n)$.

2.4.5 Pareto-criteria of Minimal Height and Minimal Cardinality

The problem of determining the set of Pareto optimal partitionings of T with respect to the criteria of minimum height and minimum cardinality is of interest, since minimizing the height and the cardinality of a partitioning are conflicting objectives, as shown by the example in Fig. 2.12. In the example, component weights are calculated as the sum of vertex weights. The vertex weights are indicated on the vertices, and the weight limit W is 10.

Below we present a polynomial time algorithm for determining the set of Pareto optimal partitionings of T with respect to three criteria: minimum height, minimum

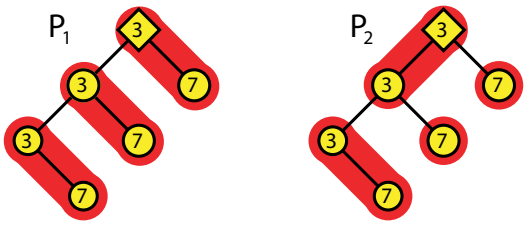


Figure 2.12: A minimal-cardinality ($q(P_1) = 3,\ h(P_1) = 2$) and a minimal-height ($q(P_2) = 4,\ h(P_2) = 1$) partitioning of the same tree, under a weight limit of 10.

cardinality and minimum root component weight. The third criterion is necessary to make the iterative bottom-up algorithm work. Note that from such a set one can easily derive the Pareto set with respect to height and cardinality. Throughout this section we assume that the component weight function w is *invariant*.

If P_v and Q_v are admissible partitionings of $T(v)$, we say that P_v *dominates* Q_v if and only if $q(P_v) \leq q(Q_v)$, $h(P_v) \leq h(Q_v)$ and $rw(P_v) \leq rw(Q_v)$. The set of Pareto optimal partitionings $PO(v)$ of a sub-tree $T(v)$ is a minimal set (w.r.t. set inclusion) of admissible partitionings of $(T(v))$ such that each admissible partitioning Q_v of $T(v)$ is dominated by some $P_v \in PO(v)$. Note that $T(v)$ may admit several different Pareto sets.

To facilitate the computation of $PO(v)$, we will use $PO^h(v)$ to denote a minimal set (w.r.t. set inclusion) of admissible partitionings of $T(v)$ such that each admissible partitioning Q_v of $T(v)$ with $h(Q_v) = h$ is dominated by some $P_v \in PO^h(v)$. Notice that $PO^h(v)$ consists of partitionings P_v with $h(P_v) \leq h$ only. Furthermore, for each q there can be at most one $P_v \in PO^h(v)$ with $q(P_v) = q$, the one whose root component weight is minimal among the partitionings of $T(v)$ that respect the height limit of h and cardinality limit of q.

Clearly, for each leaf v of T, $PO(v)$ consists of only the trivial partitioning $\{\{v\}\}$. We will show that if v is not a leaf of T, then $PO(v)$ can be constructed by combining partitionings chosen from the sets $PO(u),\ u \in S(v)$. A fundamental property of the *comb* operator is that it preserves dominance:

Lemma 2.7 *If for each $u \in S(v)$, P_u and Q_u are admissible partitionings of $T(u)$ such that P_u dominates Q_u, then $P_v = comb(\{P_u \mid u \in S(v)\}, K)$ dominates $Q_v = comb(\{Q_u \mid u \in S(v)\}, K)$, for any $K \subseteq S(v)$. Moreover, if Q_v is admissible, then so is P_v.*

Proof: For each $u \in S(v)$ we have $h(P_u) \leq h(Q_u)$, $q(P_u) \leq q(Q_u)$ and $rw(P_u) \leq rw(Q_u)$. Relating this to the equations 2.1 and 2.2, and Lemma 2.3, we obtain $h(P_v) \leq h(Q_v)$, $q(P_v) \leq q(Q_v)$ and $rw(P_v) \leq rw(Q_v)$. Hence, P_v dominates Q_v. Lemma 2.3 also implies that P_v is admissible. □

Lemma 2.8 *Let $Q_v = comb(\{Q_u \mid u \in S(v)\}, K)$ be an admissible partitioning of $T(v)$. Then there exist $P_u \in PO(u)$, $\forall u \in S(v)$ such that $P_v := comb(\{P_u \mid u \in S(v)\}, K)$ dominates Q_v.*

Proof: By the definition of the sets $PO(u)$, for each $u \in S(v)$ there exists $P_u \in PO(u)$ such that P_u dominates Q_u. Applying Lemma 2.7 to these partitionings P_u and Q_u, we can deduce that $P_v = comb(\{P_u \mid u \in S(v)\}, K)$ is an admissible partitioning of $T(v)$ dominating Q_v. □

Consequently, $PO(v)$ can be constructed by finding appropriate combinations of partitionings chosen from the sets $PO(u)$, $u \in S(v)$. We narrow subset of $PO(u), u \in S(v)$ that can participate in the construction of $PO^h(v)$ further as follows.

Lemma 2.9 *Let $P_v = comb(\{P_u \mid u \in S(v)\}, K)$ be arbitrary member of $PO^h(v)$, where $\forall u \in S(v) :\ P_u \in PO(u)$. Then the following conditions hold:*

For each $u \in K$:

(1a) $$h(P_u) \leq h$$

(1b) $$rw(P_u) = \min_{\substack{Q_u \in PO(u):\ h(Q_u) \leq h \\ q(Q_u) = q(P_u)}} rw(Q_u)$$

For each $u \in S(v) \setminus K$:

(2a) $$h(P_u) \leq h - 1$$

(2b) $$q(P_u) = \min_{Q_u \in PO(u):\ h(Q_u) \leq h-1} q(Q_u)$$

Proof: Part (1a): Suppose there exists a $u^* \in K$ with $h(P_{u^*}) > h$. Then, by equation 2.1, $h(P_v) \geq h(P_{u^*}) > h$, a contradiction.

Part (1b): Now, suppose there exists $Q_{u^*} \in PO(u^*)$ such that $h(Q_{u^*}) \leq h$, $q(Q_{u^*}) = q(P_{u^*})$ and $rw(Q_{u^*}) < rw(P_{u^*})$. Then $P'_v := comb(\{P_u \mid u \in S(v) \setminus \{u^*\}\} \cup \{Q_{u^*}\}, K)$ strictly dominates P_v, contrary to the definition of $PO^h(v)$.

Parts (2a) and (2b): Similar to parts (1a) and (1b). □

These results suggest the following method for constructing $PO^h(v)$. We define for each son $u \in S(v)$ the set $\Phi(u)$ consisting of 4-tuples

$$[P_u,\ c,\ q,\ R]$$

corresponding to options of partitioning $T(u)$ and combining it into the partitioning $P_v := comb(\{P_u \mid u \in S(v)\}, K)$ of $T(v)$. In the 4-tuple, P_u stands for a partitioning of $T(u)$, while $c = 1$ if $u \in K$ and $c = 0$ otherwise. q denotes the contribution of this option to cardinality of P_v, and can be computed as $q := q(P_u) - c$. Finally, $R \subseteq V$ is the contribution of the option to the root component of P_v. Clearly, if $c = 1$, then R equals $RC(P)$, i.e., the root component of P_u, and $R = \emptyset$ otherwise. According to Lemma 2.9, there will be two types of options in $\Phi(u)$. First,

$$[P_u,\ 1,\ q(P_u) - 1,\ RC(P_u)]$$

for each q, where $P \in PO(u)$ satisfies $h(P_u) \leq h$, $q(P_u) = q$ and $rw(P)$ smallest possible (if exists). Secondly, one option

$$[P_u,\ 0,\ q(P_u),\ \emptyset],$$

where $P_u \in PO(u)$ satisfies $h(P_u) \leq h - 1$ and $q(P_u)$ smallest possible (if exists). Now, to create a partitioning of $T(v)$, we have to choose one option $\varphi(u)$ from $\Phi(u)$ for each $u \in S(v)$. Each selection will induce a partitioning of $T(v)$

$$P_v := comb(\{P_u \mid u \in S(v)\}, K),$$

where $K = \{u \in S(v) \mid c_{\varphi(u)} = 1\}$, with the following height, cardinality and root component weight.

$$h(P_v) \leq h,$$

$$q(P_v) = 1 + \sum_{u \in S(v)} q_{\phi(u)},$$

$$rw(P_v) = w(\{v\} \cup \bigcup_{u \in S(v)} R_{\phi(u)}).$$

The partitioning P_v is admissible if and only if $rw(P_v) \leq W$ holds, because its components – except for the root component – are contained by one of its admissible generators as well.

Now, we compute $PO^h(v)$ by the following *dynamic program* [78]. Let us denote $n := |V(T(v))|$, $d := |S(v)|$ and index the elements of $S(v)$ arbitrarily: $S(v) = \{u_1, \ldots, u_d\}$. We fill in a $d \times n$ matrix, whose element $\Psi_{k,q}$ contains the selection of options that result in a minimal root weight partitioning of $T_k(v) = \{v\} \cup \bigcup_{i=1}^{k} T(u_i)$ with $h(P_v) \leq h$ and $q(P_v) \leq q$. Hence, for each $q = 1, \ldots, n$, $\Psi_{d,q}$ contains a selection corresponding to minimal root-weight partitioning P_v of $T_d(v) = T(v)$ with $h(P_v) \leq h$ and $q(P_v) \leq q$. The non-dominated partitionings corresponding to the selections $\Psi_{d,q}$ will constitute $PO^h(v)$.

For $k = 1$, let $\Psi_{1,q} := \{\varphi_q(u_1)\}$ if there exists a $\varphi_q(u_1) \in \Phi(u_1)$ with $q_1 = q$, and $\Psi_{1,q} := \emptyset$ otherwise. For $2 \leq k \leq d$, we select the option $\varphi_{q'}(u_k)$ from $\Phi(u_k)$ for which $\{\varphi_{q'}(u_k)\} \cup \Psi_{k,q-q'}$ is of the lowest root component weight, and choose

$$\Psi_{k,q} := \{\varphi_{q'}(u_k)\} \cup \Psi_{k,q-q'}.$$

Lemma 2.10 *The partitioning induced by $\Psi_{k,q}$ is of minimal root component weight among the admissible partitionings of $T_k(v)$ with $h(P_v) \leq h$ and $q(P_v) \leq q$.*

Proof: For $k = 1$, the proof directly follows from Lemma 2.9 and the definition of $\Phi(u)$. For $2 \leq k \leq d$, the root component weight of the partitionings induced by both $\varphi_{q'}(u_k)$ and $\Psi_{k,q-q'}$ is minimal among the admissible partitioning of the corresponding sub-trees, with the given height and cardinality bounds. Hence, the proof follows from the definition of the invariant property. □

Hence, we can sum up the working of the iterative step of our algorithm as follows. To compute $PO(v)$ for the selected vertex $v \in V$, it first determines the sets $PO^h(v)$ for each reasonable h (see above). Then it forms $PO(v)$ from $\bigcup_h PO^h(v)$ by dropping those members which are dominated by some other member (ties are broken arbitrarily). Since $\bigcup_h PO^h(v)$ dominates all admissible partitionings of $T(v)$ by definition, and the dominance relation is transitive, the set $PO(v)$ constructed this way is a Pareto optimal set of partitionings of $T(v)$.

Concerning the running time, the dynamic program has time complexity $O(d_v n^2)$, where $n = |V| \geq |V(T(v))| \geq |\Phi(u)|$, and $d_v = |S(v)|$. Thus $PO(v)$ can be determined in $O(d_v n^3)$ time by varying h between 1 and $|V(T(v))|$. The entire algorithm terminates in $O(n^4)$ time.

2.5 Discussion

After having presented the basic principles and properties of our aggregate production planning framework, we concern three subsidiary points that fundamentally influence the applicability of the approach. These include how activity throughput times can be estimated (Sect. 2.5.1), how a potential infeasibility or other sort of inadequacy of the solution can be mended (Sect. 2.5.2), and how the framework can be extended to cover practical needs of make-to-order industries (Sect. 2.5.3).

2.5.1 Estimating Activity Throughput Times

An essential building block of the proposed aggregation method is the ability to measure the throughput times of the activities. Since this must be carried out during aggregation, without knowing other activities processed concurrently in the factory, the measurement is inevitably based on a heuristic estimate.

Although the minimum throughput time of individual activities – with concurrent activities neglected – could be computed in theory, in practice this would put an extraordinary computational burden on the partitioning algorithms. It would require the solution of NP-complete scheduling problems each time the weight of a sub-tree is measured when partitioning the project trees. We note that minimal throughput time as a weight function is monotonous, but not invariant.

In many applications, simplistic and rapid estimators can provide a reasonable alternative for computing the minimum throughput times. When most tasks of an activity can be processed only sequentially, the sum of task durations in the activity gives a sufficiently good estimate. Similarly, if the tasks on different branches can be executed concurrently, then the sum of task durations on the longest path can be applied. Both of these weight functions, as well as their linear combinations are monotonous and invariant. In addition, they can be computed incrementally, which means that the single-criteria partitioning algorithms preserve their linear time complexity when these estimators are applied.

However, in richer scheduling models (e.g., we had to account for diverse activities built from tasks with specific transportation and setup times) the application of the latter methods results in too coarse estimations. In such cases, we suggest the use of an appropriate priority rule-based scheduler. The computational complexity of such methods is $O(mn^2)$ in general, and lower for many special cases [57]. Unfortunately, weight functions incorporating such heuristics do not hold the desirable property

of monotonicity, i.e., sometimes the addition of a tasks to an activity decreases the throughput time of the activity. This phenomenon, known as the Braess paradox [38], is illustrated in Fig 2.13.

t	d_t	$r(t)$
t_1	4	r_1
t_2	1	r_3
t_3	1	r_1
t_4	4	r_2
t_5	1	r_1
t_6	1	r_1
t_7	6	r_3
t_8	1	r_2

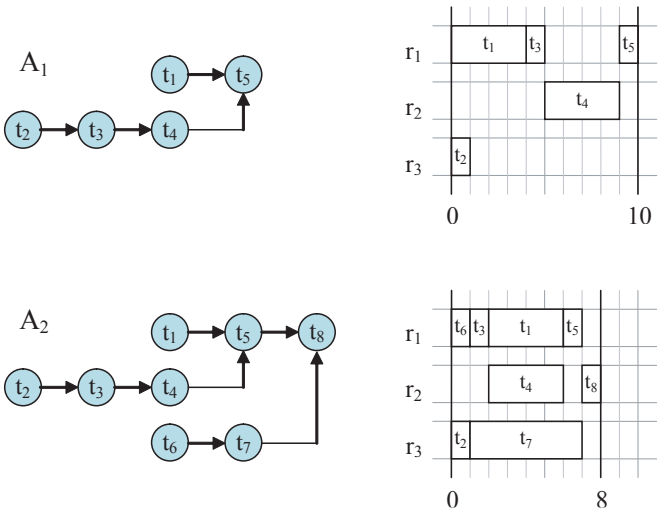


Figure 2.13: Two activities illustrating the non-monotonicity of weight functions based on priority rules. Although A_1 is a subset of activity A_2, its execution according to the LFT priority rule (see page 55) takes 10 time units, while the execution of A_2 needs only 8 time units.

2.5.2 Hand-tailoring the Production Plan

A production planning system, even if it uses the most sophisticated models and the most powerful solution algorithms, is not applicable in practice if it is unable to handle occasional requirements or preferences of production engineers. This prerequisite calls for the application of mixed-initiative techniques and the support for the hand-tailoring of the production plan. Below we suggest several ways of interaction between the PPS system and the human planner.

Obviously, the plan can be edited by *local modifications* of the model, such as setting the time windows, or even directly the intensities of individual activities. The system then should check the consistency of the manual settings and complete the draft to a complete plan. However, applying the local modifications consistently is difficult and time consuming, especially if the changes concern many activities. In

contrast, *global modifications*, such as the fine-tuning of the security factors, can handle several typical scenarios in a semi-automated way. We suggest semi-automated modifications based on the *criticality indices* of resources and activities in the specific detailed scheduling problems [4, 50, 88]. These indices measure how much a resource or an activity contributes to the objective value (e.g., makespan or maximum delay) of the given scheduling problem.

Given that the feasibility of the aggregation/disaggregation procedure cannot be guaranteed, sometimes the solutions of detailed scheduling problems do not respect every resource capacity and temporal constraint. If the capacity constraints are considered hard in the short-term scheduling problems, then this effect results in delayed tasks. Since in practice the delays will concern a few tasks only, it is likely that these tasks can be simply re-allocated to the neighboring aggregate time units. If an application requires to treat delays already on the production planning level, then delays can be resolved by decreasing the security factors of the projects and resources whose criticality index is high.

A revision of the production plan can also be motivated by preferences that are not considered in our aggregate planning model, such as levelled resource usage. Unless project time windows are too tight, peaks of oscillating load on a specific resource can be flattened by applying a lower, resource-specific security factor, or by specifying a finite extra capacity constraint. If the criticality index of the given resource is relatively low somewhere in the neighborhood of the overloaded time units, then flattening is possible without introducing new demand for extra capacities.

Finally, although we apply aggregate project models with minimal height, even the induced minimal project throughput times can be too high in the case of rush orders . These throughput times can be decreased further by applying higher, project-specific activity security factors. This can be realized without the risk of an inexecutable production plan if there are only a few rush orders.

2.5.3 Extensions and Future Research

We regard the proposed aggregation/disaggregation procedure as a *basis* for aggregate production planning in make-to-order project-oriented systems. However, different practical applications may require various extensions of the presented methods. For instance, in our industrial application we considered tasks that require *several resources* for their execution (machines and human workforce), and have specific *setup*

and *transportation times*. All this required the extension of the heuristic for the estimation of activity throughput times, and to adapt the formula for the calculation of the resource requirements of the activities. Another straightforward extension of the model is to allow aggregate time units of different length that represent, e.g., weeks containing national holidays. Such situations can be handled by time-varying intensity bounds $j_\tau^A = \min(1,\ \frac{\mu_A \Theta_\tau}{d(A)})$ on the activities.

While we assumed that all required raw material is available by the earliest start time of the projects, in some applications *raw material arrival* can be a realistic bottleneck. In such cases, aggregate project models can be optimized by inserting m pieces of dummy tasks with durations of Θ before each task whose raw materials arrive in the m^{th} aggregate time unit, see Fig. 2.14.

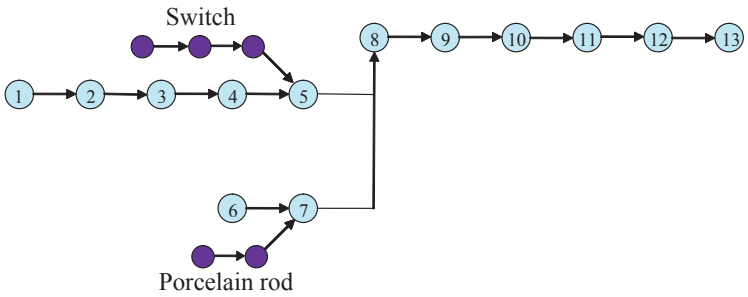


Figure 2.14: Modification of the sample project tree for the case where the switch and the porcelain rod arrive on the 3$^{\text{nd}}$ and 2$^{\text{nd}}$ time unit of the aggregate horizon, respectively.

A noteworthy extension of the aggregation procedure would be its adaption to problems where the graph of precedences among the tasks constitutes a *directed acyclic graph* (DAG), instead of an in-tree. Although such a generalization of the introduced notions is trivial, this extension induces graph partitioning problems that are NP-complete [39]. Accordingly, the construction of appropriate heuristic partitioning algorithms is inevitable.

Currently, we are working on the adaptation of the framework to manufacturing systems that carry out a combination of *project- and inventory-oriented* production. Inventory-oriented activities produce common components for project-oriented activities, which in turn, are responsible for the satisfaction of customer orders. Components are not dedicated to projects. Instead, they are treated as non-renewable resources – so-called *reservoirs* – produced or consumed by different activities.

Finally, Urgo [92] called our attention to a potential application where tasks of different projects can share setups, and the opportunity of clustering such tasks over time must not be abandoned. On the one hand, this requires the enrichment of the representation by novel relations between the activities. On the other hand, it may worth considering these relations even during aggregation, possibly through iterative adjustments.

2.6 Experiments

In order to prove the industrial applicability of the presented aggregate production planning approach, we carried out experiments on real-life production data. We investigated whether our algorithms are able to produce a compact, solvable representation of the planing problem, and if this reduced model leads to plans that can be disaggregated into feasible detailed schedules. Nevertheless, we could not measure the optimality of the aggregation procedure, since no currently known detailed scheduling approach is capable of solving such large scheduling problems to optimality. Historic data about how the projects were executed in reality was not available to us, either.

In contrast, we had access to all detailed technology and capacity related data of the factory (bills of materials, routings, and resource calendars), as well as the anticipated customer orders for a period of ca. 1 year. Departing from this data, we could generate 12 problem instances by rolling the medium-term planning horizon of 15 weeks over the one-year-long period covered by the customer orders. Between two consecutive problem instances, the timescale was rolled by 3 weeks.

Task durations in the detailed representation were specified with a precision of 0.1 hours, and each of the 1200 projects contained 20 to 500 tasks. With the choice of $\Theta = 120$ hours, which equals the number of working hours in a week, the medium-term horizon was divided into 15 aggregate time units. The activity security factor was set to 0.8, while the resource security factor varied over time: we used $\mu_R^1 = 0.8$ for the first 5 aggregate time units, $\mu_R^2 = 0.75$ for time units 6 to 10, and $\mu_R^3 = 0.7$ for the last time units of the medium-term horizon, in order to allow for unforeseen projects.

These settings led to planning problems with 600 to 900 activities, whose solution for minimum extra capacity usage required ca. 10 seconds, while the branch-and-cut search for the solution with minimal WIP was often stopped when reaching the time

limit of 100 minutes. Each aggregate solution defined 15 detailed scheduling problem instances, corresponding to the aggregate time units. The detailed problems were solved separately, by constraint-based scheduling techniques (see Chapter 3). During this, we considered resource capacity constraints hard, which implies that a potential infeasibility of the aggregation procedure manifested itself by delays of some tasks w.r.t. the end of the short-term horizon. Among the tasks we distinguished *critical tasks*, i.e., those which were executed in the last aggregate time unit of their project time window. Clearly, the delay of a critical task directly threats project deadlines, while the delay of a non-critical task is less problematic. In fact, almost the half of the tasks were critical, because there were relatively many short projects and the criterion of minimal WIP led to allocating the activities near to the latest finish time of the corresponding project. All in all, we solved each detailed scheduling problem in two steps: first, the maximum delay of the critical tasks, and then the maximum overall delay was minimized. The time limit of the scheduler was set to 10 minutes, which was not always enough to find an optimal solution.

	Aggregate planning				Detailed scheduling							
					Total				Critical			
	Tasks	Activities	Planned	Broken	Tasks	Delays	Avg. Delay	Max. Delay	Tasks	Delays	Avg. Delay	Max. Delay
#1	43213	2413	621	42	9277	67	5.72	16.7	6035	19	3.85	11.7
#2	42850	2358	753	56	12127	83	6.28	25.3	7713	23	3.99	12.1
#3	42140	2247	833	49	13196	89	6.46	19.7	8554	27	5.29	17.2
#4	38255	2074	826	53	13824	75	6.11	17.6	5530	14	3.89	13.6
#5	34637	1901	801	37	13120	60	8.09	29.2	3294	5	4.50	11.5
#6	32549	1756	855	35	14536	105	8.37	45.4	4344	43	5.77	12.8
#7	30249	1584	847	32	16024	83	7.99	50.4	3930	30	3.71	14.1
#8	27892	1397	770	32	15308	140	9.00	45.4	3917	87	8.23	18.6
#9	24271	1244	769	22	15346	74	7.32	41.8	5871	16	2.95	15.6
#10	21068	1073	720	30	14490	165	7.64	35.0	6144	62	5.67	14.8
#11	16412	871	702	31	14001	151	5.33	22.5	5492	61	4.36	10.3
#12	13956	726	722	38	13804	160	5.04	20.9	7051	82	3.95	8.5

Table 2.2: Results on a set of industrial problems.

The results are presented in Table 2.2. Each row of the table corresponds to one problem instance, i.e., one aggregate planning and the 15 induced detailed scheduling problems. The overall number of tasks in the problem instance is indicated in the first column, the number of activities generated from them in the second. Hence, an activity contained ca. 20 tasks on average. However, only a part of the activities

were allocated within the medium-term horizon, as shown in column *Planned*. Since a part of the resource capacity constraints was tight in every time period, ca. 5% of the activities were broken in every aggregate solution, see column *Broken*.

Finally, the portion of delayed tasks in the detailed solutions was between 0.5% and 1.1%. The average delay (over the tasks whose delay was other than 0) was around 5.0 - 9.0 hours, which is about the 6% of Θ. However, the delays of a few tasks was significantly larger, the greatest delay exceeded even 50 hours. As for the critical tasks, the 0.1% - 1.0% of them were delayed, their average delay was 2.95 - 8.23 hours, while the maximal critical delay reached between 8.5 and 18.6 hours.

Despite we did not manage to achieve complete feasibility of the aggregation, we believe that these results are competitive with the performance of any currently known production planning methods, and our approach definitely outperforms the widely used MRP techniques. In addition, even the above delays can be eliminated by common sense methods, e.g., by exchanging a few tasks between neighboring aggregate time units. Small delays can be simply disregarded when the latest finish time of a project is set earlier than the customer shipment date. If these are not possible, then delays pointed out in time can be dissolved by hiring extra capacities, e.g., overtime. On the other hand, shop-floor disturbances can often cause significantly more delays. Such situations were analyzed by discrete-event simulation, and will be discussed in detail in Sect. 4.6.

2.7 Conclusions

In this chapter, we have emphasized the role of aggregation as the primarily link between the models of production planning and scheduling. We regarded aggregation as a representation problem, and demonstrated that its solution has a significant impact both on the quality of the production plans and the feasibility of the detailed schedules. Hence, proper aggregation is a major prerequisite for using advanced PPS methods successfully.

We have presented a novel aggregation/disaggregation framework that – with suitable extensions – can serve as a flexible and efficient basis for production planning in make-to-order production environments. The experimental results achieved on real-life production data support this claim. The fact that the aggregate model of production planning can be generated automatically, from master data readily

available in de facto standard production information systems is of vital practical significance.

Finally, we note that the proposed linear-time single-criteria and polynomial-time bi-criteria partitioning algorithms may find applications in many other fields as well.

Chapter 3

Consistency Preserving Transformations in Constraint-based Scheduling

The solution of complex practical scheduling problems requires flexible representation and efficient solution methods. Constraint-based scheduling, that relies on the declarative programming paradigm of constraint programming, is an attractive approach in both aspects [10].

While constraint solvers offer a declarative way to represent most conceivable features of detailed scheduling problems, the constraint-based approach is also regarded as the most efficient exact approach for solving various classes of resource-constrained project scheduling problems. Thanks to the above characteristics, constraint-based scheduling is more than just an intensively studied area of theoretical research. It has also taken pride in numerous practical applications by now. What is more, today's most wide-spread enterprise information systems incorporate constraint-based schedulers [79, 84].

However, real-life industrial problems often require richer models and contain an order of magnitude more tasks than typical scheduling benchmarks used by research communities. The complexity and size of these industrial problems challenge even the most advanced constraint-based scheduler systems, and current algorithms often fail to solve the problem instances to an acceptable range of the optimum. Our goal was to improve the performance of constraint-based scheduling techniques by exploring and exploiting common structural properties of industrial scheduling problems [63, 64].

In this chapter, we first review existing representations, algorithms, and current research directions in constraint-based scheduling (Sect. 3.1). Then, we propose

several novel methods that belong to the class of *consistency preserving transformations* to improve the performance of the previously described algorithms on practical scheduling problems (Sect. 3.2). Results of tests carried out on real-life problem instances are also presented in the same section. Finally, in Sect. 3.3, we draw the conclusions and indicate some interesting directions for future research.

3.1 Introduction to Constraint-based Scheduling

3.1.1 Representation of the Scheduling Problem

A constraint program Π is defined by a 4-tuple $\{X, D, C, O\}$ as follows. $X = \{x_i\}$ denotes a finite set of *variables*. Each variable x_i can take a value from its *domain* D_i. There is a set of *constraints* C defined on the variables. The set of variables present in the N-ary constraint $c(x_{i_1}, \ldots, x_{i_N}) \in C$, or briefly c, is denoted by $X_c = \{x_{i_1}, \ldots, x_{i_N}\}$. Then, the solution of a constraint program is a binding S of the variables, i.e., $\forall x_i \in X : x_i = v_i^S \in D_i$ such that all the constraints are satisfied, $\forall c \in C : c(v_{i_1}^S, \ldots, v_{i_N}^S) = true$. The last member of the 4-tuple, O stands for an *objective function* which orders a real number to a solution S. If $O \equiv 0$, i.e., an arbitrary solution of the constraint program is looked for, then we call Π a *constraint satisfaction problem*, otherwise, we talk about a *constraint optimization problem*. In the remainder of this chapter, we will assume that optimization means the minimization of O.

The resource-constrained project scheduling problem, as defined in Section 2.2.1, can be formulated within this framework as follows. The variables are the start times $start_t$ of the tasks $t \in T$. The durations d_t are assumed to be integers, and consequently, the initial domain of every start time variable is the set of integers from 0 to a sufficiently large number.[1] These domains are later tightened by the constraints. In order to simplify the description of the forthcoming algorithms, we also introduce the following notions. The functions $Dmin$ and $Dmax$ return the minimum and maximum of the given variable's domain, respectively.

- The earliest start time of a task t, $est_t = Dmin(start_t)$;
- The latest start time of t, $lst_t = Dmax(start_t)$;
- The earliest finish time of t, $eft_t = Dmin(start_t) + d_t$;

[1] Some scheduling models assign three variables, $start_t$, d_t, and end_t to each task, addressing problems where durations are not known in advance.

- The latest finish time of t, $lft_t = Dmax(start_t) + d_t$;
- If $start_t$ is bound, then the end time of t will be denoted by $end_t = start_t + d_t$;
- The time window of t is the interval $[est_t, lft_t]$.

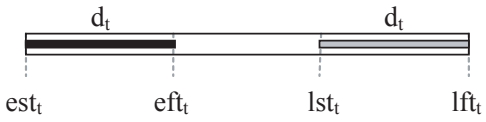


Figure 3.1: The time window of a task.

For an illustration of the above notions, see Fig. 3.1. Now, we are looking for such a binding of the variables $start_t$ that satisfies different temporal, precedence and resource constraints.

- *Temporal constraints* state earliest start or latest finish times for tasks, e.g., $start_t \geq \tau$. They are *unary*, i.e., concern only one variable, and can be translated into a domain reduction immediately.
- *Precedence constraints* are binary, and state that the tasks t_1 and t_2 have to be executed in the given order. An *end-to-start* precedence constraint between t_1 and t_2 requires that $end_{t_1} \leq start_{t_2}$, while a start-to-start precedence constraint only prescribes $start_{t_1} \leq start_{t_2}$. These will be denoted by $(t_1 \rightarrow t_2)$ and $(t_1 \dashrightarrow t_2)$, respectively, and determine a directed acyclic graph over the set of the tasks together. Note that although the description of RCPSP contains only end-to-start precedence constraints, we will infer additional start-to-start precedences during the solution process.
- Finally, *resource capacity constraints* ensure that the total of the resource requirements of the tasks processed concurrently never exceeds the available capacity. A resource capacity constraint is always responsible for *one* resource r, and concerns *all* the tasks which require this resource. *Unary resources*, i.e., resources with $q(r) = 1$, are often distinguished from *cumulative resources* $(q(r) > 1)$. Scheduler systems often provide constraints to represent *reservoirs*, i.e., resources that can be consumed or produced by tasks, and so-called *state resources* that are able to process a task only when they are in a given state, such as a furnace which can be heated to different temperatures.

The most common objective function is minimizing the makespan, i.e., the maximum of the end times $end_t, t \in T$. Other maximum-type objective functions, such as peak resource usage, or, when individual due dates are specified for the projects, maximum tardiness can be minimized efficiently in a very similar way, too. Although most other usual optimization criteria can also be expressed in constraint languages, constraint-based scheduling is less efficient on those problems.

In order to illustrate constraint-based models of scheduling problems, we present the encoding of the well-known job-shop scheduling problem in the OPL constraint language [94] in Fig. 3.2. A problem instance is presented in Fig. 3.3, while Fig. 3.4 shows an optimal solution for the same instance through the project view of a Gantt chart.

```
int nMachines = ...;
int nJobs = ...;
int nTasks = ...;

int duration[1..nJobs, 1..nTasks] = ...;
int resource[1..nJobs, 1..nTasks] = ...;

Activity task[j in 1..nJobs, t in 1..nTasks] (duration[j,t]);
UnaryResource machine[1..nMachines];
var int makespan in scheduleOrigin..scheduleHorizon;

minimize
  makespan
subject to
{
  //Temporal constraints:  all the tasks end before the makespan
  forall(j in 1..nJobs)
    task[j,nTasks].end <= makespan;

  //Precedence constraints between successive tasks of a job
  forall(j in 1..nJobs)
    forall(t in 1..nTasks-1)
      task[j,t] precedes task[j,t+1];

  //Resource constraints
  forall(j in 1..nJobs)
    forall(t in 1..nTasks)
      task[j,t] requires machine[resource[j,t]];
};
```

Figure 3.2: Encoding of the job-shop scheduling problem model in the OPL language.

```
nMachines = 6;
nJobs = 6;
nTasks = 6;
resource = [
  [ 3, 1, 2, 4, 6, 5],
  [ 2, 3, 5, 6, 1, 4],
  [ 3, 4, 6, 1, 2, 5],
  [ 2, 1, 3, 4, 5, 6],
  [ 3, 2, 5, 6, 1, 4],
  [ 2, 4, 6, 1, 5, 3]
    ];
duration = [
  [ 1, 3, 6, 7, 3, 6],
  [ 8, 5, 10, 10, 10, 4],
  [ 5, 4, 8, 9, 1, 7],
  [ 5, 5, 5, 3, 8, 9],
  [ 9, 3, 5, 4, 3, 1],
  [ 3, 3, 9, 10, 4, 1]
    ];
```

Figure 3.3: Description of the ft06 job-shop problem instance in the OPL language.

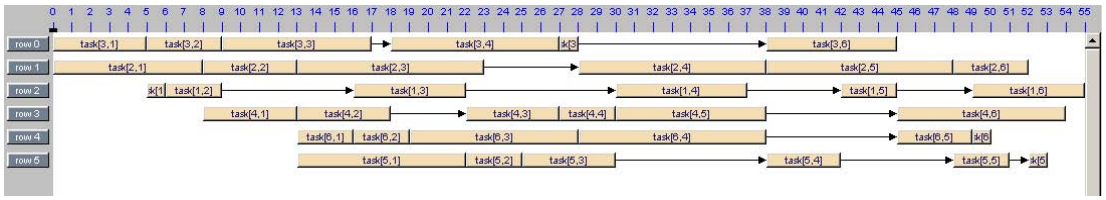


Figure 3.4: An optimal solution of the ft06 job-shop problem instance. Excerpt from the output of Ilog OPL Studio.

3.1.2 Transformations of Constraint Programs

The solution process of a constraint program generally consists of a tree search. Since in most applications the targeted problems are hard, or more specifically, NP-complete, computational efficiency is a crucial issue. Constraint programming earns this efficiency from the *transformations* performed on the constraint problem. These transformations, in general, produce an easier-to-solve representation of the problem, e.g., by tightening the variables' domains. According to the definitions in [2], a transformation $\Pi \Rightarrow \Pi'$ is called *equivalence preserving* if for every binding S of the variables, S is a solution of Π iff it is also a solution of Π'.

However, a wider set of transformations, the so-called *consistency preserving* transformations are eligible to solve problems when one has to decide only whether

Π has a solution or not. A transformation $\Pi \Rightarrow \Pi'$ is defined to be consistency preserving, if it holds that Π' has a solution iff Π has a solution.

Indisputably, the most important transformation technique in constraint programming is *constraint propagation*. A propagation algorithm is always attached to a particular constraint c, and addresses tightening the domains of the variables in X_c by removing the values which are provably inconsistent with c. Sometimes, the propagator is also able to infer new, implied constraints, and adds them to the constraint program. Clearly, the removal of a value from the domain of a variable can render some members of other variable domains infeasible according to another constraint. This can result in a chain effect. The propagation chain ends at a fix point when no propagators can eliminate further inconsistent values. Propagation is executed in each node of the search tree, performing an equivalence preserving transformation. Below, we briefly overview the algorithms that can be applied to propagate constraints in scheduling.

Simple temporal and precedence constraints can be propagated easily by a standard arc-B-consistency algorithm [72]. Arc-B-consistency is a type of interval consistency, i.e., the propagators adjust the lower and the upper bounds of the variables' domains, while they do not consider the inner values. For example, the simple temporal constraint $start_t \geq \tau$ results in the domain reduction $est'_t = \max\{est_t, \tau\}$. A precedence constraint $t_1 \rightarrow t_2$ can tighten the time window of both t_1 and t_2: $lft'_{t_1} = \min\{lft_{t_1}, lft_{t_2} - d_{t_2}\}$ and $est'_{t_2} = \max\{est_{t_2}, est_{t_1} + d_{t_1}\}$, see Fig. 3.5.

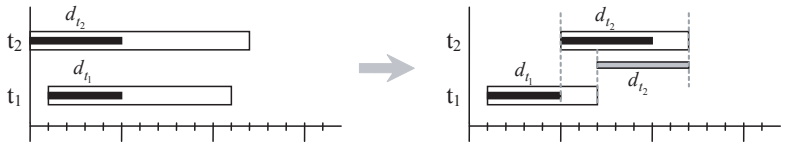


Figure 3.5: Propagating the $t_1 \rightarrow t_2$ precedence constraint.

Propagating resource constraints is a more challenging problem, because it contains a nested one-machine scheduling problem which is NP-complete. Consequently, polynomial-time propagation algorithms cannot guarantee that they make all the possible domain reductions. In what follows, $T(r)$ will always denote the set of tasks to be processed on the unary resource r, amongst which, obviously, at most one can be processed at a time. The simplest algorithm to propagate the unary resource

constraint is *time-tabling* . It considers tasks one by one, and if for some $t \in T(r)$ it finds that $eft_t < lst_t$, then it deduces that in the time interval $[eft_t, lst_t]$ t and only t is executed. Hence, the time windows of the tasks in $T(r) \setminus \{t\}$ can be tightened accordingly, see Fig. 3.6.

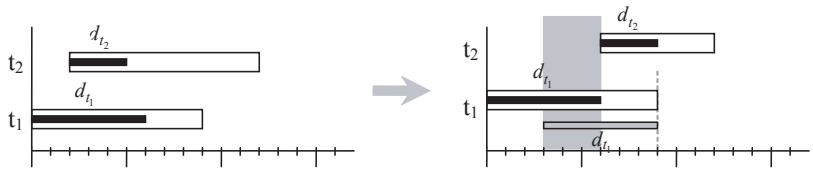


Figure 3.6: Applying the time-tabling algorithm.

The *disjunctive* propagation algorithm [29] consists of maintaining arc-B-consistency on the formula $(t_1 \rightarrow t_2) \vee (t_2 \rightarrow t_1)$ for each pair of tasks $t_1, t_2 \in T(r)$. Whenever $lst_{t_1} < eft_{t_2}$ occurs, i.e., $t_2 \rightarrow t_1$ proves false, the algorithm deduces $t_1 \rightarrow t_2$ and makes the corresponding domain tightenings, and vice versa.

The introduction of the so-called *interval consistency tests*, such as the *edge-finding algorithm* [21] or the *not-first, not-last* test [89] was a breakthrough in constraint-based scheduling, both in computational efficiency and in a theoretical aspect. They infer new time bounds and precedence constraints by the global comparison of resource requirement and capacities in different time intervals. Below, we present the framework suggested in [19] for the description of these tests. The general idea is based on the observation that given two task sets $U_1, U_2 \subseteq U$, such that $U \subseteq T(r)$, if

$$lft_{max}(U \setminus U_2) - est_{min}(U \setminus U_1) < \sum_{t \in U} d_t,$$

then a member of U_1 must start first or a member of U_2 must end last in U. The proof of this statement can be easily read from Fig. 3.7 which shows the timescale divided into three intervals. The intervals are delimited by $est_{min}(U \setminus U_1)$ and $lft_{max}(U \setminus U_2)$. Then, only members of U_1 can be processed at interval I, and only members of U_2 in interval III. Although *any* tasks can be executed in interval II, the interval is too short to process *all* the tasks in U, hence, either in interval I or in interval III, some work must be done.

Since U_1 and U_2 can be selected from $T(r)$ in exponentially many ways, only special cases of the statement are investigated in practice. Table 3.1 summarizes the commonly used interval consistency tests.

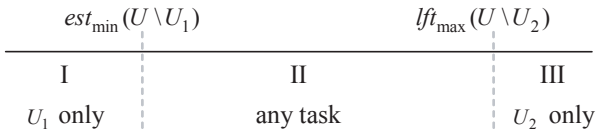


Figure 3.7: The basic idea of interval consistency tests.

Name of the test	U_1	U_2	Conclusion
Input	$\{t_1\}$	$\emptyset$	t_1 executes first in U
Output	$\emptyset$	$\{t_1\}$	t_1 executes last in U
Input-or-output	$\{t_1\}$	$\{t_2\}$	t_1 is first or t_2 is last in U
Input negation	$T \setminus \{t_1\}$	$\{t_1\}$	t_1 is *not* first in U
Output negation	$\{t_1\}$	$T \setminus \{t_1\}$	t_1 is *not* last in U

Table 3.1: Interval consistency tests.

The edge-finding algorithm, which performs equivalents of the input and the output tests, is extremely efficient in solving job-shop type scheduling problems, and it is incorporated in all current constraint-based scheduler systems. The working of the algorithm is illustrated in Fig. 3.8 on 3 tasks. We note that there exists an implementation of the edge-finding algorithm with $O(n \log n)$ time complexity [22], but in practice, the $O(n^2)$ implementation described in [7] generally runs faster thanks to the simpler data structures applied.

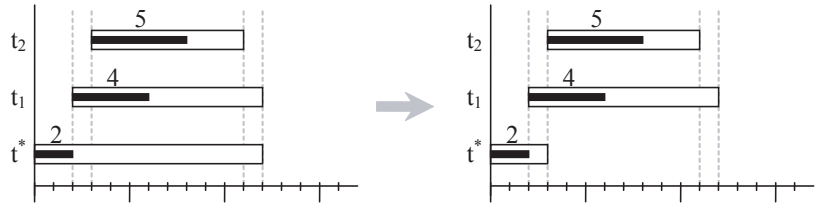


Figure 3.8: Application of the edge-finding algorithm.

Although several researchers attempted to strengthen further the propagation on unary resources w.r.t. the above interval consistency tests, these experiments generally achieved only a little more domain tightening at the cost of significantly more computing time invested, see, e.g., [99]. Hence, it is widely accepted today that

no significant improvement can be expected on the above propagation algorithms *in the case of unary resources*.

However, the situation fundamentally differs for cumulative resources, reservoirs, and state resources . Although the above propagation algorithms can be generalized to cumulative resources [9], in fact, they achieve significantly weaker pruning. A stronger propagation algorithm with time complexity $O(n^3)$, called *energetic reasoning* has been suggested in [31]. It compares, in appropriately selected time intervals, the total amount of work required by the tasks on the given resource to the available capacity.

Two further algorithms, the *energy precedence* and the *balance constraint propagators* are described in [67]. These algorithms, unlike the above propagators that compute domain tightenings based on the time windows of the tasks, focus on the precedence relations between them. They are remarkably more efficient than the previous propagators for cumulative resources and reservoirs if the tasks' time windows are wide.

Shaving is another equivalence preserving transformation that is widely used in scheduling applications, when constraint propagation itself is unable to achieve sufficient search space reduction [90]. Shaving adds an arbitrary constraint c to the constraint program Π, and if propagation algorithms prove $\Pi \cup \{c\}$ infeasible – which is the lucky case here –, then it infers that $\neg c$ must be fulfilled in all the solutions of Π. In scheduling, c typically stands for a time bound on a task's start/end time or a precedence constraint.

3.1.3 Search Techniques

Besides the above transformations that reduce the search space, the way of exploring this search space is also decisive for the efficiency of the solution process. In this section, we focus on search issues.

Constraint optimization problems are solved through a reduction of the optimization problem to a series of satisfiability problems. In successive search runs, the feasibility of the problem is checked for different trial values of the objective function. In the case of a *branch-and-bound search*, the objective value of the actually best solution, i.e., an upper bound UB is stored, and the solver looks for a solution with an objective value of at most $UB - 1$. If such a solution is found, UB is updated, while infeasibility means that the previous solution was an optimal one. The advantage of the branch-and-bound search is that search does not have to be restarted

from scratch, but it can be continued from the given node of the search tree with an updated upper bound.

In contrast, during a *dichotomic search*, the solver keeps in mind both the actually known best upper bound UB and lower bound LB, i.e., the lowest value for which infeasibility has not been proven. Then, in each search run, the trial value $\lfloor (UB + LB)/2 \rfloor$ is probed, and, depending on the outcome of the trial, either the value of UB or LB is updated. This step is repeated until $UB = LB$ is reached, which means that an optimal solution has been found. Although dichotomic search restarts the solution process in each successive run, it makes larger "jumps" towards the optimum in the initial phase of the solution process. Whether the branch-and-bound or the dichotomic search is worth applying depends on the specific problem.

Within each optimization step, the search trees are generally explored in a depth-first order. This strategy is explained by the high memory needs of constraint solvers. In addition, constraint solvers apply an incremental description of the search nodes, i.e., they do not store all the data connected to the given node, but only the changes compared to the parent node. This makes switching between two distant nodes of the tree expensive. Consequently, heuristics are rarely exploited in sophisticated informed search methods, but rather in the smart selection of the search decisions within the search nodes.

Limited discrepancy search, an alternative to the depth-first search was introduced in [48]. It is based on the assumption that if a good heuristic misses a solution, then it is due to only a small number of bad search decisions. Hence, it defines *discrepancy* as the number of branchings on a search path in which a decision different from the one suggested by the heuristic was made. Then, limited discrepancy search divides the search tree into strips, each corresponding to the set of nodes with 0, 1, 2, etc. discrepancies, and it explores the strips in this order.

There is also a choice of the types of decisions to make at the search nodes. *Resource ranking* and *task pair ordering* are the most widely used branching schemes when unary resources are scheduled. Resource ranking selects a resource r and a task $t \in T(r)$, and generates a binary branching according to the decision wether t is the next task on r or not. Task pair ordering selects a pair of tasks t_1, t_2 to be processed on the same resource and branches on $t_1 \rightarrow t_2$ or $t_2 \rightarrow t_1$.

In both of the above branching schemes, it is beneficial to follow the *fail-first* principle [47], which states that search should focus on making the critical choices first. In scheduling, this means that the most loaded resource has to be ranked or the

most tight pair of tasks has to be ordered first. Sophisticated, so-called *texture-based heuristics* are suggested in [14] to identify the critical decisions in job-shop scheduling problems. Similar analysis methods, named *profile-based metrics* are suggested in [23] that are tailored to cumulative resource models. Alternatively, a clique-based approach is proposed by [68] to find the subsets of tasks whose resource requirements can produce conflict.

Although the previous branching schemes can also be generalized to cumulative resources, their cumulative counterparts can only add start-to-start precedence constraints to the model, because several tasks can be processed concurrently on the same resource. These weaker precedence constraints often cannot trigger sufficient domain tightenings. Instead, the so-called *setting times* branching scheme is applied. This strategy relies on the LFT priority rule [26], and binds the start times of tasks in the following way. It first selects the earliest time instant τ for which there exists a non-empty set T_τ of unscheduled tasks that can be started at time τ. A task $t \in T$ belongs to T_τ iff all its end-to-start predecessors have ended and all its start-to-start predecessors have started by τ, and there are enough free units of the resource $r(t)$ in the interval $[\tau, \tau + d_t]$. From T_τ, the task t^* with the smallest latest finish time lft_{t^*} is selected. The setting times branching scheme then generates two sons of the current search node, according to the decisions whether $start_{t^*}$ is bound to est_{t^*}, or t^* is postponed.

However, complete tree search methods sometimes poorly scale to large-size problems. This phenomenon initiated extensive research on combining the inference power of constraint propagation with the better scalability of local search techniques. The price payed for the computational efficiency of these hybrid methods is loosing their completeness. Nevertheless, current state of the art in this field rather consists of pieces of experience gathered during individual experiments, than a well-established methodology. A comprehensive overview of the existing approaches can be found in [35]. Herein, we discuss only two frameworks that can guarantee the feasibility of the solutions.

One of the efficient generic methods is to exploit constraint programming in exploring a larger neighborhood by a b ranch-and-bound search within each iteration of the local search. This approach is called *large neighborhood search* [80]. An application of this approach to solve the job-shop scheduling problem is presented in [8]. In this case, in each iteration of the local search, each ordering decision of the previously found schedule is kept with a given probability p, while others are relaxed. Then, a

constraint-based depth-first search is run to find a solution which improves on the best known makespan, within a limited number of backtracks. This step is iterated with a decreasing p, until a certain terminating condition is reached.

A completely different approach that searches in the space of consistent partial solutions, the so-called *incomplete dynamic backtracking*, was introduced in [81]. It starts the solution process with an empty set of variable assignments. Then, in each iteration, an unassigned variable is selected and bound to a value in its domain. Then, domains of the other variables are tightened by constraint propagation. If any of the domains becomes empty – which means that the current set of assignments cannot be completed to a solution –, then one or more heuristically selected assignments are undone. This step is iterated until all variables are assigned, which means that a solution is found. [54] presents how this framework can be applied to open-shop scheduling, a problem which is notoriously hard for exact methods.

3.1.4 Application Problems of Constraint-based Scheduling

The above described techniques made constraint programming an attractive representation and solution method for solving complex, real-life scheduling problems. There are quite a number of companies selling scheduling software based on constraint technology. Despite all this, in many applications it turns out that even the most advanced systems are often unable to solve large problems – which may include an order of magnitude more tasks than typical benchmarks used by researchers – to an acceptable range of the optimum. At the same time, it has also been recognized that practical problems can often be simple in the sense that they have a certain internal structure. We believe that the key to success in solving real-life problems is the conscious exploitation of these structural properties.

This requirement has been recognized in several connected fields of combinatorial problem solving. As a result, SAT (boolean satisfiability problem) solvers released during the last couple of years are capable of solving one order of magnitude larger problems than their predecessors. Many of them exploit the *backdoors* of problems, i.e., a set of variables such that if they are bound to the appropriate values, the original problem simplifies to a trivial one, e.g., a 2-SAT [98]. The authors report that they found backdoors consisting of 10 to 20 variables for problems containing 10^5 variables. Another typical structure in a SAT problem that can be exploited during search is the so-called *backbone* of the problem. It is the set of variables which must take the same values over all solutions [85].

Although the need for focusing on more realistic problems has been a common claim also in constraint programming research, we do not have a clear understanding of what kind of problem structure can be expected and exploited in constraint programs, or, more specifically, in constraint-based scheduling problems.

A further discrepancy between theoretical research and practice involves the idea of optimization. Researchers most often aim at finding the *exact optimum* according to some relatively easy-to-handle criteria, such as the makespan in scheduling [13]. In contrast, engineers can typically settle for a *sufficiently good* solution, but they call for a wider choice of more realistic objective functions. Moreover, they often measure the quality of a solution according to several criteria, but researchers cannot offer sophisticated tools for *multi-criteria optimization*. E.g., the Pareto branch-and-bound search suggested in [62, 75] has a clear theoretic background, but can be applied efficiently only when the set of Pareto-optimal solutions is small enough.

In our research, we aimed at identifying structural properties of constraint-based scheduling problems that originated from a real factory. We defined two generic approaches – consistency preserving transformations – to exploit these features of the problems so as to make search more efficient. These transformations will be presented in the following sections. At the same time, we still considered makespan as the optimization criterion. This was reasonable, because the main cost drivers were located at the medium-term planning level of our integrated PPS system, while short-term scheduling was responsible for realizing the objectives set by the medium-term plan.

3.2 Consistency Preserving Transformations for the Exploitation of Problem Structure

The transformations of constraint programs presented so far all preserve equivalence. In fact, the reason for current general purpose constraint solvers to perform equivalence preserving transformations is rooted in their modular structure. They incorporate local inference algorithms, such as propagators attached to individual constraints. These algorithms do not have a view of the entire model, and hence, they can remove only such values from the variable domains which cannot be part of any solution because violate the given constraint. In contrast, transformations which do not preserve equivalence, remove also values which *can* participate in some of the solutions. Without loosing the chance of finding a solution (or proving infeasibility),

this is possible only with an overall, global view of the model.

Nevertheless, we claim that consistency preserving transformations are adequate means to exploit problem structure in specific application domains. Furthermore, it is possible to construct generic consistency preserving transformation frameworks that can be easily tailored to particular problems. Specifically, we can observe that the following structural properties are often present in real-life industrial scheduling problems [64].

- Many factories produce *families of products*, members of which share sections of their technological plans. Moreover, several orders for the same product can be present at the factory within the scheduling horizon. All this results in a number of similar or even identical sub-problems within the scheduling problem.

- Members of a product family use a common set of resources, while, on the other way around, different product families often require basically different (though not disjoint) sets of resources. The projects visit those resources in sequences more or less determined by the manufacturing technology applied.

- The amount of load often significantly differs over resources, and hence, there are *bottleneck* and *non-bottleneck resources*. Similarly, there can be *critical* and *non-critical* projects. The latter properties typically result in a scheduling problem that consists of a loosely connected structure of easy and hard sub-problems.

In what follows, we first present some currently applied consistency preserving transformations (Sect. 3.2.1). Then, we suggest two novel consistency preserving transformations. The first one exploits the presence of similar sub-problems in the scheduling problem (Sect. 3.2.2). Then, Sections 3.2.3 and 3.2.4 present how loosely connected sub-problems can be exploited in constraint programming and specifically in constraint-based scheduling, respectively. Finally, experiments on real-life data and also on widely used benchmark instances are presented in Sect. 3.2.5.

3.2.1 Related Work

Recently, several efforts have been made to explore consistency preserving transformations in constraint programming. Typical transformations which preserve consis-

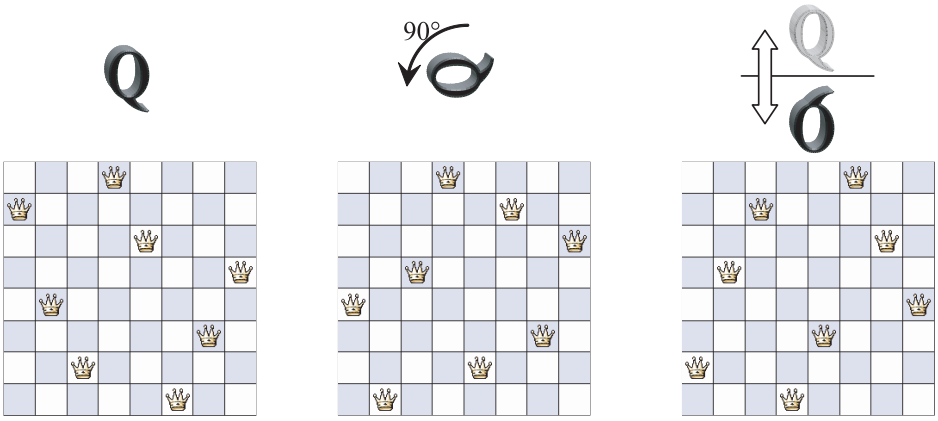


Figure 3.9: Symmetric solutions of the 8-queens puzzle.

tency, but do not retain equivalence, are the applications of *symmetry breaking* and *dominance rules*.

Symmetry is a structural property of constraint satisfaction problems. It is a bijective function f defined on the bindings of the variables of a constraint program Π such that for each variable binding α, $f(\alpha)$ is a solution of Π iff α is a solution, too. Clearly, a problem can have several, even an exponential number of symmetries in the function of the number of variables. Classical examples of symmetry are the rotational and reflection symmetries in the 8-queens puzzle, as illustrated in Fig. 3.9. In this problem, eight queens have to be placed on a chessboard in such a way that none of them attacks the others.

The presence of symmetries in a problem can give rise to redundant search on equivalent branches of the search tree. Consequently, breaking the symmetries of constraint programs, i.e., excluding all but one of the equivalent solutions is an extensively studied research area today. The forefather of all symmetry breaking techniques is the addition of symmetry breaking constraints to the model *before search* [24]. For instance, row and column symmetries in matrix models can be eliminated by lexicographical ordering constraints [34]. The flaw of this simple approach is that it hardly scales up to break a high number of symmetries, and that it can interact adversely with search heuristics.

More sophisticated methods, such as the Symmetry Breaking During Search [42] (also called Symmetry Excluding Search in [6]) and the Symmetry Breaking via Dominance Detection [32, 36] prune symmetric branches of the search tree *during search*.

All of these general frameworks require an explicit declaration of the symmetries in the form of symmetry functions or a dominance checker. Recently, these algorithms were improved by lessons learnt from computational group theory. These improvements eliminate the requirement of specifying a huge number of symmetry functions, and make it possible to break all the symmetries of a problem by using only the description of the generators of symmetry groups. The resulting algorithms are called GAP-SBDS [40] and GAP-SBDD [41].

A wider class of consistency preserving transformations is constituted by the *dominance rules*. They define properties of a problem that must be satisfied by at least one of its (optimal) solutions. By now, little work has been done to explore domain independent dominance rules. E.g., a recent paper [82] introduced the notion of *k-dominance consistency* to provide means for the removal of such values from the variable domains whose infeasibility can be proven from the dead ends met in earlier stages of the search. However, the authors define algorithms only for restricted cases, and the applicability of this rather complicated method can hardly be understood yet.

At the same time, domain specific dominance rules are widely used in constraint programming, as well as in resource-constrained project scheduling. For instance, two similar dominance rules are suggested in [7, 27] that bind the start time of a task to the earliest possible value if its predecessors are already processed and the given resource is not required by any other task at that time. A dominance rule to decompose the scheduling problem over time is described in [7]. More complex – and more expensive – dominance rules are discussed by [28]. Several dominance rules as well as rules for the insertion of redundant precedence constraints are proposed for the problem of minimizing the number of late jobs on a single machine, see [11].

In what follows, we suggest two novel equivalence preserving transformations that – according to the above classification – belong to the family of dominance rules. However, in contrast to the dominance rules outlined above, they perform global inference on the constraint program. The first technique partially orders the tasks of the scheduling problem by the insertion of precedence constraints before search. The second algorithm can be run in each node of the search tree to build partial solutions that are provably consistent with all the solutions of the remaining part of the problem. The precursors of these techniques were first described in [63].

3.2.2 Progressive Solutions of Scheduling Problems

In the previous sections we have seen that real-life scheduling problems often contain similar sub-problems corresponding to the production of several identical products or members of the same product family. Here, we suggest a method for the deduction of a part of the ordering decisions between corresponding tasks of such similar projects *before search*. These inferred ordering decisions allow the insertion of precedence constraints in the problem representation, and thus, make possible a significant reduction of the search space. Since these investigations are performed before search on the initial representation of the scheduling problem, in the following definitions we do not take into consideration the notions only used within the constraint-based solver, such as time windows and start-to-start precedence constraints. We begin by formally characterizing similarities between two task sets.

Definition 3.1 *Two sets of tasks P and Q are defined isomorphic, and will be denoted by $P \equiv Q$, iff there exists a bijection $\beta : P \leftrightarrow Q$ such that for each pair of tasks $p \in P$ and $q \in Q$*

$$\begin{aligned} \beta(p,q) &\Rightarrow\ d_p = d_q \wedge r(p) = r(q), \text{ and} \\ \beta(p_1,q_1) \wedge \beta(p_2,q_2) &\Rightarrow\ (p_1 \rightarrow p_2) \Leftrightarrow (q_1 \rightarrow q_2). \end{aligned}$$

Definition 3.2 *A set of tasks $P \subseteq T$ is called closed iff it is a maximal connected component in the graph of precedences of T.*

Definition 3.3 *Given two closed task sets P and Q, we call them a progressive pair iff there exists a $P^* \subseteq P$ and a $Q^* \subseteq Q$ such that $P^* \equiv Q^*$, and there are no incoming precedences to P^* and no outgoing precedences from Q^*. This relation will be denoted by $P \rightrightarrows Q$ (see Fig. 3.10.a).*

Note that the closed task sets in a scheduling problem are defined unambiguously. The progressive pair relations are also determined unambiguously, unless for two closed task sets P and Q, $P \equiv Q$ holds, which means that the direction of the progressive pair relation between them can be chosen arbitrarily. In such cases, we select the directions so that no directed circles of progressive pair relations are constructed. Furthermore, if the bijection β (and, accordingly, P^* and Q^*) can be chosen in several ways within a progressive pair, then we fix one of the alternatives containing a maximal number of tasks. This way, the progressive pair relations determine an acyclic partial ordering of the closed task sets.

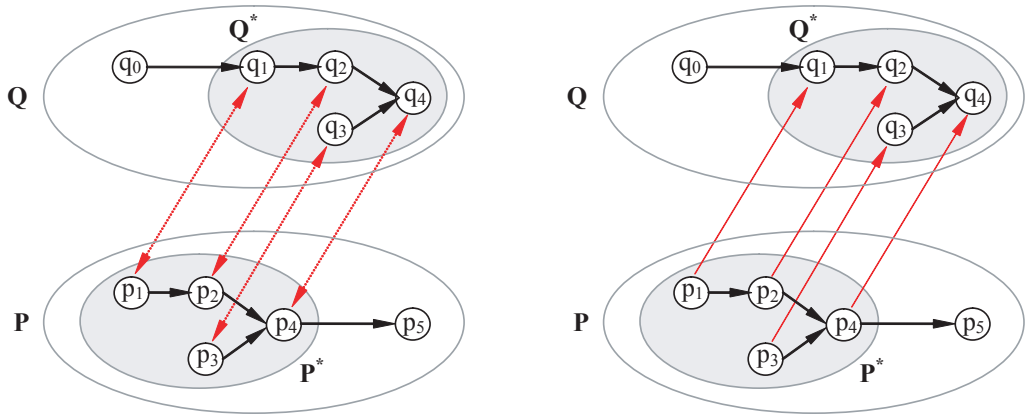


Figure 3.10: a.) The progressive pair $P \rightrightarrows Q$. b.) The progressive precedence constraints between P and Q.

Definition 3.4 *A solution of a scheduling problem is called progressive, iff for each progressive pair $P \rightrightarrows Q$, the execution of P precedes Q, in the formal sense that for each pair of tasks $p \in P^*$ and $q \in Q^*$ such that $\beta(p,q)$, $p \dashrightarrow q$ holds. We will refer to this type of start-to-start precedence constraints as progressive constraints (see Fig. 3.10.b).*

We note that if the resource $r(p)$ is unary, then $(p \dashrightarrow q) \Leftrightarrow (p \rightarrow q)$.

Theorem 3.1 *If a scheduling problem has a solution, then it also has a progressive solution.*

We will prove this theorem by an algorithm that departs from an arbitrary solution of the scheduling problem, and through iteratively swapping pairs of tasks, generates a progressive solution. In each step of the algorithm, a progressive pair $P \rightrightarrows Q$ is selected, such that some of the progressive constraints between P and Q are violated in the actual schedule S. Then, the algorithm computes a modified schedule S' by swapping all the pairs of tasks in P and Q which violate the progressive constraints as follows.

$$\forall p \in P, q \in Q : \beta(p,q) \wedge start_p^S > start_q^S \quad \Rightarrow \quad start_p^{S'} = start_q^S, \text{ and } start_q^{S'} = start_p^S.$$

For all other tasks $t \in T$, $start_t^{S'} = start_t^S$.

Lemma 3.1 *S' is feasible.*

Proof: All resource capacity constraints are satisfied in S', because only pairs of tasks with equal durations and resource requirements were swapped. In order to show that precedence constraints $p_1 \rightarrow p_2$, where $p_1, p_2 \in P^*$, cannot be violated in S' either, we introduce q_1 and q_2 to denote the two tasks in Q for which $\beta(p_1, q_1)$ and $\beta(p_2, q_2)$ hold. Then,

- if neither the pair (p_1, q_1), nor (p_2, q_2) were swapped, then the start times of p_1 and p_2 are unchanged in S' w.r.t. S, and S is feasible;
- If the pair (p_1, q_1) was swapped, but (p_2, q_2) not, then $end_{p_1}^{S'} = end_{q_1}^{S} < end_{p_1}^{S} \leq start_{p_2}^{S} = start_{p_2}^{S'}$;
- If the pair (p_2, q_2) was swapped, but (p_1, q_1) not, then $end_{p_1}^{S'} = end_{p_1}^{S} \leq start_{q_1}^{S} \leq start_{q_2}^{S} = start_{p_2}^{S'}$;
- If both (p_1, q_1) and (p_2, q_2) were swapped, then $end_{p_1}^{S'} = end_{q_1}^{S} \leq start_{q_2}^{S} = start_{p_2}^{S'}$.

Precedence constraints pointing from P^* to $P \setminus P^*$ and those within $P \setminus P^*$ are also satisfied, because only tasks of P^* were moved forward, and tasks of Q^* backward in the schedule. The proof is analogous for precedence constraints in Q, and trivial for the precedence constraints between tasks of $T \setminus (P \cup Q)$, because those tasks were not moved. □

The above step is iterated until there are no more progressive constraints violated.

Proof of Theorem 3.1: The algorithm halts when it has found a progressive schedule. According to Lemma 3.1, this schedule is feasible, too. Furthermore, this is reached in finitely many steps, because the algorithm performs a brick sort over the closed task sets, according to the partial ordering defined by the progressive pair relations. Hence, we have shown that the insertion of the progressive constraints according to Definition 3.4 preserves the consistency of the scheduling problem. □

In our system, the set of tasks contained by one aggregate activity constitute a closed task set (see Chapter 2). The progressive pair relation $P \rightrightarrows Q$ characterizes two activities belonging to projects that share – at least partially – their technological plans, and P can be in a somewhat more advanced state than Q. Finding the progressive pairs and composing the corresponding progressive precedence constraints

is straightforward. For each pair of activities, we check whether their contained task sets constitute a progressive pair. Since the precedence graph of an activity forms an in-tree structure, this can be performed by examining if one of the activities contain the other activity's root task. If it does, then the corresponding sub-trees must be mapped to each other.

3.2.3 Freely Completable Partial Solutions

Broadly speaking, a *freely completable partial solution* (FCPS) is a consistent binding of a subset of the variables, which does not constrain the domain of the remaining variables in any way. FCPSs are traits of such constraint satisfaction problems that have some components which are relatively easy to solve and are only loosely connected to the remaining parts of the problem. Once detected, freely completable partial solutions can be exploited well during the solution process: search decisions in the easy-to-solve sub-problems can be eliminated, and search can be focused on making the relevant decisions only.

In what follows, we define a general, domain independent framework to identify freely completable partial solutions. The framework can be applied to boost search in constraint satisfaction problems and constraint optimization problems which are solved as a series of satisfiability problems. Hence, in the formal definitions, we will use the notations introduced in Sect. 3.1.1. We will prove that the transformation we suggest preserves consistency. Later, in Sect. 3.2.4, we will show how this generic framework can be exploited to boost search efficiency in constraint-based scheduling.

A partial solution PS is a binding of a subset $X^{PS} \subseteq X$ of the variables, $\forall x_i \in X^{PS} : x_i = v_i^{PS}$. We define PS freely completable, iff for each constraint $c \in C$:

- If $X_c \subseteq X^{PS}$, then $c(v_{i_1}^{PS}, \ldots, v_{i_N}^{PS}) = true$, i.e., c is satisfied.
- If $X_c \nsubseteq X^{PS} \wedge X_c \cap X^{PS} \neq \emptyset$, then let $D'_{i_k} = \{v_{i_k}^{PS}\}$ for $x_{i_k} \in X^{PS}$, and $D'_{i_k} = D_{i_k}$ for $x_{i_k} \notin X^{PS}$. Then, $\forall (u_{i_1}, \ldots, u_{i_N}) \in D'_{i_1} \times \ldots \times D'_{i_N} : c(u_{i_1}, \ldots, u_{i_N}) = true$. Note that this means that *all the possible* bindings of the variables not included in PS lead to the satisfaction of c.
- If $X_c \cap X^{PS} = \emptyset$, then we make no restrictions.

Freely completable partial solutions are illustrated in Fig. 3.11. In the example, PS is an FCPS if the constraint $c1$ is satisfied in PS, and $c2$ is satisfied for all the

possible bindings of its variables in $X \setminus X^{PS}$. The constraint $c3$, no matter how hard-to-solve sub-problem it defines, does not have to be cared about.

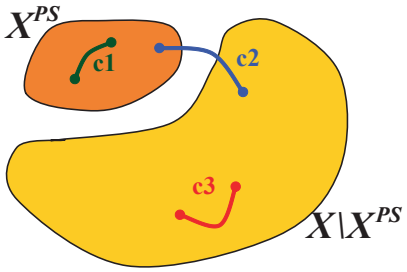


Figure 3.11: Example of a freely completable partial solution.

Theorem 3.2 *If PS is a freely completable partial solution, then binding the variables $x_i \in X^{PS}$ to the values v_i^{PS}, respectively, is a consistency preserving transformation.*

Proof: Suppose that there exists a solution S of the constraint program. Then, the preconditions in the above definition prescribe that the binding $x_i \in X^{PS} : x_i = v_i^{PS}, x_i \notin X^{PS} : x_i = v_i^S$ is also a solution, because it satisfies all the constraints. On the other hand, it is trivial that any solution of the transformed problem is a solution of the original problem, too. □

Note that whether a partial solution is freely completable or not, depends on *all* the constraints present in the model. In case of an optimization problem, this includes the constraints posted on the objective value as well. Thus, this transformation can not be applied e.g., within a branch-and-bound search, where such constraints are added during the search process.

A freely completable partial solution PS, apart from the trivial $X^{PS} = \emptyset$ case, does not necessary exist for constraint satisfaction problems, or finding one can be just as difficult as solving the original problem. Notwithstanding, we claim that in structured, practical problems, fast and simple heuristics are often capable to generate such a PS. In what follows, this will be demonstrated for the case of constraint-based scheduling.

3.2.4 Application of FCPSs in Constraint-based Scheduling

A partial solution PS of a scheduling problem is a binding of the start time variables $start_t$ of a subset of the tasks, which will be denoted by $T^{PS} \subseteq T$. According to the

previous definitions, PS is called freely completable, if the following conditions hold for each constraint of the model.

For end-to-start precedence constraints $c : (t_1 \rightarrow t_2)$,

- $t_1, t_2 \in T^{PS}$ and $end_{t_1} \leq start_{t_2}$, i.e., c is satisfied, or
- $t_1 \in T^{PS}, t_2 \notin T^{PS}$ and $end_{t_1} \leq est_{t_2}$, i.e., c is satisfied irrespective of the value of $start_{t_2}$, or
- $t_1 \notin T^{PS}, t_2 \in T^{PS}$ and $lft_{t_1} \leq start_{t_2}$, i.e., c is satisfied irrespective of the value of $start_{t_1}$, or
- $t_1, t_2 \notin T^{PS}$, i.e., PS does not make any commitments on the start times of t_1 and t_2.

This definition can be extended to start-to-start precedence constraints $c : (t_1 \dashrightarrow t_2)$ likewise:

- $t_1, t_2 \in T^{PS}$ and $start_{t_1} \leq start_{t_2}$, or
- $t_1 \in T^{PS}, t_2 \notin T^{PS}$ and $start_{t_1} \leq est_{t_2}$, or
- $t_1 \notin T^{PS}, t_2 \in T^{PS}$ and $lft_{t_1} - dt_1 \leq start_{t_2}$, or
- $t_1, t_2 \notin T^{PS}$.

To check resource capacity constraints, we define $M^+_{r,\tau}$ as the set of tasks $t \in T^{PS}$ which are under execution at time τ on resource r, while $M^-_{r,\tau}$ as the set of tasks $t \notin T^{PS}$ which *might be* under execution at the same time:

$$M^+_{r,\tau} = \{t | t \in T^{PS} \wedge r(t) = r \wedge (start_t \leq \tau \leq end_t)\}$$
$$M^-_{r,\tau} = \{t | t \notin T^{PS} \wedge r(t) = r \wedge (est_t \leq \tau \leq lft_t)\}$$

Now, one of the followings must hold for every resource $r \in R$ and for every time unit τ:

- $|M^+_{r,\tau}| + |M^-_{r,\tau}| \leq q(r)$, i.e., the constraint is satisfied at time τ irrespective of how PS will be complemented to a complete schedule, or
- $M^+_{r,\tau} = \emptyset$, i.e., PS does not make any commitment on r at time τ.

A Heuristic Algorithm

We applied the following heuristic algorithm to construct freely completable partial solutions of scheduling problems. The algorithm can be run once in each search node, with actual task time windows drawn from the constraint solver.

The method is based on the LFT priority rule-based scheduling algorithm [26], which also serves as the origin of the setting times branching strategy. It was modified so that it generates freely completable partial schedules when it is unable to find a consistent complete schedule. The algorithm assigns start times to tasks in a chronological order, according to the priority rule, and adds the processed tasks to T^{PS}.

```
1 PROCEDURE FindAnyCaseConsistentPS()
2     % Let U be the set of tasks not yet scheduled.
3     U := {t|t ∈ T : start_t is not bound};
4     WHILE (U ≠ ∅)
5       Choose a task t ∈ U and a start time τ using the LFT rule;
6       Remove t from U;
7       IF τ + d_t ≤ lft_t THEN
8         start_t := τ;
9         Add t to T^PS;
10      ELSE
11        FailOnTask(t);

12 PROCEDURE FailOnTask(t)
13    IF t ∈ T^PS THEN
14      Remove t from T^PS;
15    FORALL task t' ∈ T^PS : (t' → t) ∈ C
16      IF end_t' > est_t THEN
17        FailOnTask(t');
18    FORALL task t' ∈ T^PS : (t' --→ t) ∈ C
19      IF start_t' > est_t THEN
20        FailOnTask(t');
21    FORALL task t' ∈ T^PS : r(t') = r(t)
22      % Let I be the time interval in which t and t' can be
23      % processed concurrently.
24      I := [start_t', end_t'] ∩ [est_t, lft_t];
25      IF ∃τ ∈ I : |M+_r(t),τ| + |M-_r(t),τ| > q(r(t)) THEN
26        FailOnTask(t');
```

Figure 3.12: The heuristic algorithm for constructing freely completable partial schedules.

Whenever the heuristic happens to assign an infeasible start time to a task t, i.e., $start_t > lft_t - d_t$, t is removed from T^{PS}. The removal is recursively continued on

all tasks t' which are linked to t by a precedence or a resource capacity constraint, and whose previously determined start time $start_{t'}$ can be incompatible with any value in the domain of $start_t$. After having processed all the tasks, the algorithm returns with a freely completable partial schedule PS. In the best case, it produces a complete schedule, $T^{PS} = T$, while in the worst case, PS is an empty schedule, $T^{PS} = \emptyset$. The pseudo-code of the algorithm is presented in Fig. 3.12.

Certainly, this simple heuristic can be improved in many ways. First of all, we applied a small random perturbation on the LFT priority rule. This leads to slightly different runs in successive search nodes, which allows finding freely completable partial solutions which were missed in the ancestor nodes. In experiments (see Sect. 3.2.5), the modified rule, named LFT^{rand}, resulted in roughly 20% smaller search trees than LFT.

The time spent for building potentially empty partial schedules can be further decreased by restricting the focus of the heuristic to partial schedules PS which obviate the actual branching in the given search node. Task t^*, whose immediate scheduling or postponement is the next search decision in the constraint-based solver, is already known before running the heuristic. This next branching would be eliminated by PS only if $t^* \in T^{PS}$. Otherwise, finding PS does not immediately contribute to decreasing the size of the search tree, and it is likely that PS will only be easier to find later, deeper in the search tree. Accordingly, when `FailOnTask` is called on t^*, the heuristic algorithm can be aborted and an empty schedule returned. These improvements can be realized by replacing one line and adding three lines to the pseudo-code of the basic algorithm, as shown in Fig. 3.13.

```
1 PROCEDURE FindAnyCaseConsistentPS()
...
5     Choose a task t ∈ U and a start time τ using the LFT^rand rule;
...

12 PROCEDURE FailOnTask(t)
12A   IF t is the task on which the branching is anticipated THEN
12B     T^PS := ∅;
12C     EXIT; % The next branching cannot be avoided.
...
```

Figure 3.13: Improvements of the heuristic algorithm.

An Illustrative Example

In the following, an example is presented to demonstrate the operation of the heuristic algorithm that constructs freely completable partial schedules. Suppose there are 3 projects, consisting of 8 tasks altogether, to be scheduled on three unary resources. Tasks belonging to the same project are fully ordered by end-to-start precedence constraints. The durations and resource requirements of the tasks are indicated in Fig. 3.14, together with the time windows received by the heuristic algorithm from the constraint-based solver in the root node of the search tree. The trial value of the makespan is 10.

Note that in order to be able to present a compact but non-trivial example, we switched off the edge-finding resource constraint propagator in the constraint solver engine, and used the time-tabling propagator only.

t	d(t)	est_t	lft_t	r(t)
t11	1	0	2	R3
t12	4	1	10	R1
t21	2	0	3	R3
t22	2	2	5	R2
t23	5	4	10	R3
t31	2	0	3	R2
t32	4	2	7	R1
t33	3	6	10	R2

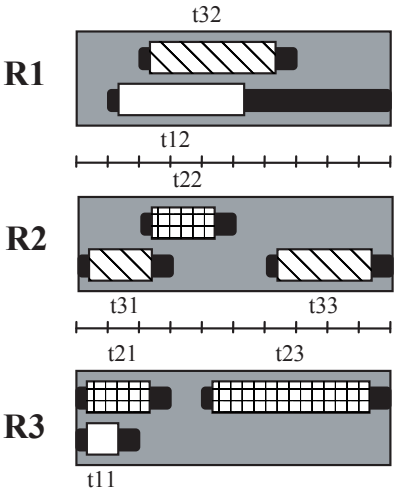


Figure 3.14: Parameters of the sample problem.

The algorithm begins by assigning start times to tasks in chronological order, according to the LFT priority rule: $start_{t11} = 0$, $start_{t31} = 0$, $start_{t21} = 1$, $start_{t12} = 1$ and $start_{t22} = 3$, see Fig. 3.15.a. All these tasks are added to T^{PS}.

Now, it is the turn of $t32$. Unfortunately, its execution can start the soonest at time 5, and consequently, it cannot be completed within its time window. Hence, the function `FailOnTask` is called on $t32$, and recursively on all the tasks which could cause this failure. At this example, it only concerns $t12$ which is removed from T^{PS}. Then, further tasks are scheduled according to the LFT priority rule: start times are

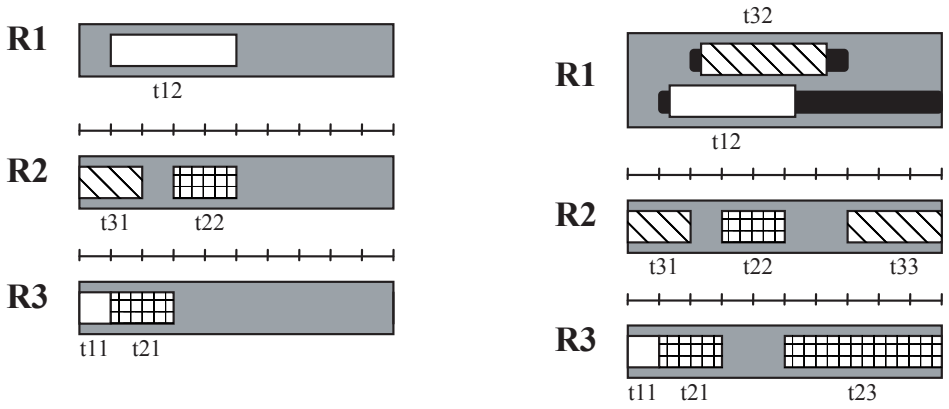


Figure 3.15: a.) Building the partial schedule. b.) The freely completable partial schedule.

assigned to the two remaining tasks, $start_{t23} = 5$ and $start_{t33} = 7$. The heuristic algorithm stops at this point, and it returns the freely completable partial schedule PS with $T^{PS} = \{t11, t21, t22, t23, t31, t33\}$, see Fig. 3.15.b.

After having bound the start times of these tasks in the constraint-based solver, the solver continues the search process for the remaining two tasks. In the next search node, it infers the only remaining valid start times for $t12$ and $t32$ by propagation. This leads to an optimal solution for this problem, as shown in Fig. 3.16.

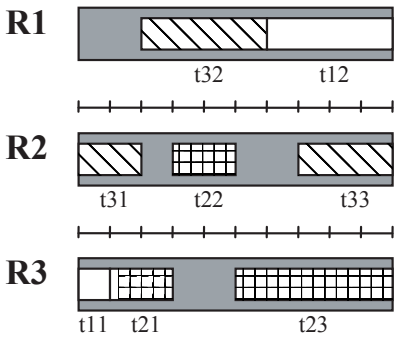


Figure 3.16: The final schedule.

3.2.5 Experiments

The purpose of our experiments was to appraise how much the exploitation of problem structure, or more specifically, the transformations and algorithms introduced

previously in this chapter can speed up the solution of scheduling problems. Below we present results achieved on real-life scheduling problem instances and widely used benchmark problems.

Search Strategy

The suggested algorithms were implemented as extensions to a state of the art commercial constraint-based scheduler system, Ilog Scheduler [52]. Since the default branch-and-bound solution approach of this system performed poorly on the real-life problem instances we addressed, it was replaced by a dichotomic search. The motivation for this change was the observation that the optimal solution of our industrial problems is often located near to the lower bound which can be proven by pure propagation. Moreover, rather surprisingly, the satisfiability problems for makespans around the optimal value – deciding whether there exists a schedule whose makespan is at most the given value – were easier to solve than for larger values of the objective function. Fig. 3.17 shows a qualitative picture of problem complexity, measured in the average number of search nodes, as a function of the trial value of the makespan.

The 'double dichotomic' search method we used first looks for the best lower bound which can be proven by propagation only. Then, while searching for a solution, it focuses its efforts on the area close to the lower bound. The algorithm is presented in Fig. 3.18.

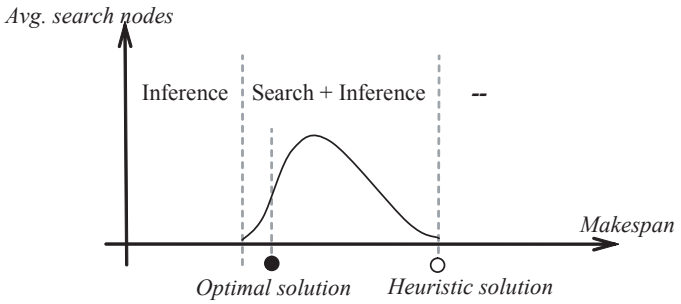


Figure 3.17: Problem complexity as a function of the makespan.

Within the second phase of the double dichotomic search, a standard depth-first search with the setting times branching strategy is performed to check if a schedule exists with at most the given makespan. During search, the time windows of the tasks are tightened by constraint propagators. For propagating precedence constraints,

```
1 PROCEDURE Schedule()
2     % Generate an initial solution.
3     S := InitialSolution();
4     UB := GetMakespan(S);
5
6     % Perform a dichotomic search to find a good lower bound
7     UB_0 := UB, LB_0 := 0;
8     WHILE (UB_0 ≠ LB_0)
9       IF propagation proves the makespan ⌊(UB_0 + LB_0)/2⌋ infeasible THEN
10        LB_0 := ⌊(UB_0 + LB_0)/2⌋;
11      ELSE
12        UB_0 := ⌊(UB_0 + LB_0)/2⌋;
13
14    % Solve the problem by a dichotomic search
15    LB := LB_0, MS := LB + 1;
16    WHILE (UB ≠ LB)
17      IF a solution with makespan at most of MS exists THEN
18        S := Solution(MS);
19        UB := MS;
20      ELSE
21        LB := MS;
22      MS := lfloor(UB + 10 * LB)/11⌋;
23
24    RETURN(S);
```

Figure 3.18: The double dichotomic search method.

the ordinary arc-B-consistency algorithm, while for resource capacity constraints the edge-finding algorithm is applied.

This solver was extended by a pre-processor that adds the progressive precedence constraints to the model, and the heuristic algorithm for constructing freely completable partial schedules, run once in each search node. Both extensions were encoded in C++. The experiments were executed on a 1.6 GHz Pentium IV computer, under a Windows 2000 operating system.

Results on Industrial Problems

Our set of industrial test instances consists of weekly detailed scheduling problems. We constructed these test instances by unfolding production plans generated by the medium-term planner module of our integrated production planner and scheduler. The input of the medium-term planner directly originated from our industrial partner.

The products of this enterprise can be ordered into four product families. A project, aimed at the fabrication of one end product, usually contains 50 to 500 ma-

chining, assembly and inspection operations. These operations were merged into 1 to 10 aggregate activities. The horizon of the short-term scheduling problem covered the execution of one or more such activities of each project. In the experiments presented below, we supposed tasks requiring one machine resource for their execution, and we also disregarded some modelling features, such as setup and transportation times. Finally, the problem instances contained ca. 80 unary and 10 cumulative resources. Further characteristics of the problems and more details about the production processes at the factory can be found in Sect. 4.1.

The industrial problems were divided into two sets. Problem set 1 consists of 30 detailed scheduling problems that our system faced on the short-term level, each containing from 150 up to 990 tasks. Problem set 2 contains 20 larger problem instances with up to 2021 tasks, generated by merging several problems from set 1.

Four systems participated in the tests: DD denotes the double dichotomic search using only built-in propagators of the commercial solver. First, DD was extended by the pre-processor to reduce the search space to progressive solutions (DD+PS), then by the algorithm for building freely completable partial solutions (DD+FC). In the last system, all components were switched on (DD+PS+FC). The solution time limit was set to 120 seconds.

Although these problems were hard for the default branch-and-bound search of the constraint-based scheduler, even the simplest algorithm, DD could find optimal solutions for all but one member of problem set 1. Reducing the search space to progressive solutions further improved on the results, but the systems exploiting freely completable partial solutions were the definite winners, thanks to an extremely low number of search nodes. In many cases, including those where the first solution proved to be optimal, these two systems could solve the problems without any search. The results are presented in Table 3.2, with separate rows for instances which could be solved to optimality (+) and those which could not (–). *Search time* and *search*

Method	Number of instances	Avg. search nodes	Avg. search time (sec)	Avg. Error (%)
DD (+)	29	282.5	2.00	-
DD (–)	1	59073.0	120.00	12.0
DD+PS (+)	30	272.1	1.67	-
DD+FC (+)	30	8.0	0.83	-
DD+PS+FC (+)	30	6.6	0.73	-

Table 3.2: Results on problem set 1.

nodes both include finding the solutions and proving optimality. *Error* is measured by the difference of the best known upper and lower bounds, in the percentage of the lower bound.

Although problem set 2 contained significantly larger problems, 14 of them were solvable by the standard methods of DD, as presented in Table 3.3.[1]. Just like on problem set 1, identifying the freely completable partial solutions of the problems remarkably reduced the size of the search tree. This way, the complete system could solve 4 further problem instances that remained unsolvable for DD. We emphasize that even among these large problems, there were some which could be solved by pure inference, without any search.

At the same time, still poor results on unsolved instances show that while these algorithms could extend the applicability of constraint-based scheduling techniques, they by themselves cannot provide the desired scalability for these systems. Scalability, indeed, could be achieved by more sophisticated search techniques, e.g., local search algorithms. However, our experiments carried out with such methods, namely two similar tabu search based RCPSP solvers proposed in [5] (without any constraint propagation) show that the size of these problem instances can be challenging even for local search techniques: both versions had to quit search because they ran out of memory on every member of problem set 2.

Results on Benchmark Instances

The systems were also tested on Lawrence's job-shop benchmark problems la01-la10 [12]. Since progressive pairs are not present in these instances, and they basically lack the loosely connected structure of industrial problems, too, we did not expect the complete system to significantly improve on the performance of the commercial constraint-based scheduler. In fact, it turned out that smaller freely completable partial solutions also exist in these benchmark instances, and our algorithms managed to decrease the size of the search tree by an order of magnitude, but this reduction did not always return the time invested in the construction of freely completable partial schedules. We note that for the instance la08, the number of search nodes is higher when using the freely completable partial solutions because the two systems depart from different initial solutions. The results are presented in Table 3.4.

[1] An extended set of problem instances is available online at `http://www.mit.bme.hu/~akovacs/projects/fcps/instances.html`.

	Tasks	DD			DD+PS			DD+FC			DD+PS+FC		
		Nodes	Time (sec)	Error (%)	Nodes	Time (sec)	Error (%)	Nodes	Time (sec)	Error (%)	Nodes	Time (sec)	Error (%)
#1	836	836	14	-	836	11	-	0	8	-	0	0	-
#2	1027	1027	21	-	2054	22	-	0	11	-	0	1	-
#3	1138	2280	27	-	2276	27	-	13	11	-	0	10	-
#4	944	18650	120	7.1	12547	120	4.2	30	14	-	9	8	-
#5	1328	10294	120	11.0	9779	120	4.2	382	120	2.0	9	13	-
#6	639	24991	120	12.2	13785	65	-	2083	120	0.8	8	5	-
#7	1141	12334	120	8.9	2283	32	-	730	120	4.2	137	30	-
#8	994	1988	21	-	1988	22	-	0	7	-	0	8	-
#9	1932	3864	101	-	3857	110	-	0	22	-	0	24	-
#10	1876	3745	99	-	3745	106	-	18	28	-	81	55	-
#11	2021	2021	76	-	2021	82	-	0	30	-	0	33	-
#12	1637	1637	46	-	1637	50	-	0	20	-	0	23	-
#13	1771	1771	53	-	1771	59	-	0	24	-	0	24	-
#14	1337	4004	45	-	1337	27	-	794	112	-	212	32	-
#15	1592	3175	52	-	3184	55	-	525	106	-	0	16	-
#16	1098	1098	32	-	1098	40	-	0	18	-	73	29	-
#17	953	953	22	-	953	28	-	0	14	-	6	13	-
#18	819	819	17	-	811	22	-	0	11	-	0	13	-
#19	1218	12630	120	17.1	4390	120	6.4	878	120	14.4	828	120	7.0
#20	1165	7559	120	10.9	4370	120	6.4	977	120	14.4	890	120	7.0

Table 3.3: Results on problem set 2.

	Size	DD			DD+FC		
		Nodes	Time (sec)	Error (%)	Nodes	Time (sec)	Error (%)
la01	10x5	92	2	-	10	0	-
la02	10x5	9489	11	-	1196	6	-
la03	10x5	149	0	-	10	0	-
la04	10x5	11390	11	-	234	1	-
la05	10x5	50	0	-	22	0	-
la06	15x5	75	0	-	2	0	-
la07	15x5	154	0	-	59	0	-
la08	15x5	144	0	-	278	2	-
la09	15x5	75	0	-	8	0	-
la10	15x5	27568	120	24.1	0	0	-

Table 3.4: Results on Lawrence's benchmarks.

3.3 Conclusions

In this chapter we suggested general notions and specialized algorithms to boost the performance of current solvers on constraint satisfaction problems that have an internal structure. We found the means for the exploitation of this structure in equivalence preserving transformations.

The introduced concepts were put into practice in the field of resource-constrained project scheduling. Specifically, we suggested a method to restrict the search space to progressive solutions by adding inferred precedence constraints to the model, when there are many similar task sets to be executed within the scheduling horizon. We also presented a procedure that addresses problems that have easy-to-solve and loosely connected sub-problems in their internal structure. We argued that solutions of such components should be discovered and separated as freely completable partial solutions by a consistency preserving transformation.

The algorithms were validated on large-size practical scheduling problems, where they showed that even in a problem containing a huge number of variables, often only a few search decisions really matter. Such problems are hard to solve for pure propagation-based solvers because many search decisions produce equivalent choices. However, by constructing freely completable partial solutions we were able to avoid growing the search tree by branchings on irrelevant search decisions, and thus scheduling problems of large size became tractable. At the same time, the experiments also demonstrated that strong inference methods cannot completely substitute sophisticated search techniques, and the latter are inevitable for constraint solvers to scale up to large practical problems as well.

Finally, we note that making the generic framework of freely completable partial solutions operational in other application areas of constraint programming is possible, and only requires the creation of heuristic algorithms that build freely completable partial solutions for the given problem class.

Chapter 4

A Pilot Production Planner and Scheduler System

The last chapter of the thesis focuses on the practical applicability of our theoretical results described in the foregoing. It presents how the aggregate planner and the constraint-based scheduler can be adjusted to the needs of real-life industries and fit together into an integrated production planner and scheduler system.

Although our system in general addresses make-to-order project-oriented manufacturing systems, the problems were specified and the applied models were worked out with the cooperation of one specific industrial partner. Real-life test problem instances also originate from this company, a Hungarian plant of a multinational enterprise that manufactures complex mechanical products for the energy industry.

Beyond basic research, the scope of this project also covered the demonstration of the industrial applicability of the theoretical results through a software prototype. The developed integrated production planner and scheduler system was named Proterv-II (Projekt tervező – project planner, in Hungarian). Since it is a pilot system for experimental and demonstration purposes, it lacks many functionalities substantial for industrial software, e.g., advanced reporting, hand-tailoring and troubleshooting options. Nevertheless, it operated on the data directly received from the company's enterprise information system, and its algorithms were successfully tested on these real-life problem instances. At the time of writing this thesis, our industrial partner shows interest towards the system, but it is still an open issue whether an extended version of Proterv-II will find application at that factory.

This chapter is organized as follows. In Sect. 4.1, we outline the current production environment at the targeted field of our application. Afterwards, in Sect. 4.2 we define the goals of the production planner and scheduler system. Then, in Sect. 4.3

we briefly overview the tools we used for the development of Proterv-II, as well as the components of the factory's information systems that we could rely on as data sources. Afterwards, we discuss in detail the models and algorithms applied in the production planner (Sect. 4.4) and the production scheduler (Sect. 4.5) sub-systems. Finally, in Sect. 4.6, we investigate how the results of our mathematical models can bear up under the uncertainties and data inaccuracy or unavailability of a realistic production environment.

4.1 Production Environment at the Target Enterprise

The target enterprise manufactures mechanical products of high value for the energy industry. Its entire production addresses the fulfillment of customer orders, i.e., the factory is a typical make-to-order production environment. An order always refers to a single product, which belongs to one of the four product families. Each family contains product variants with the same structure but different parameters. Although products from different families significantly differ in their production technology, the problem cannot be decomposed, because the sets of resources used for producing them are not disjoint. At present, the overall number of different products is ca. 40, but this number may grow in the future.

Figure 4.1: A product of the enterprise.

The *master production schedule* specifies a *time window*, i.e., a release date (earliest start time) and a deadline (latest finishing time) for fulfilling each accepted order on the *medium-term horizon*. The length of this horizon ranges from three to six months, and the time unit is one week. Since raw material arrival is sometimes erratic, the enterprise endeavors to acquire all the raw material of a product before the release date of the project. On the other hand, deadline observance is an absolute must, even for unpredicted orders.

Each product is made in bundles called sets. The size of the set is fixed for each

product since the complete set will be built into the same larger unit. Members of a set are indistinguishable, and the production of a set is not divisible by forming smaller groups.

As usual, the assembly structure of a product is given by its *bill of materials* (BOM). The BOM is tree-structured. It is rooted at the end product, while its leaves stand for the raw materials that are purchased from outside of the factory. Raw materials and intermediate products are all dedicated to specific projects, even if some of them may be technically equivalent. Consequently, there is no need to perform material planning.

To each intermediate product in the BOM tree, there is a sometimes lengthy sequence of *operations* assigned. This operation sequence is specified in the *routings*. A typical project consists of 20 to 500 discrete manufacturing operations altogether, with operation processing times in the range of 0.5 to a maximum of 120 hours. Subsequent operations cannot overlap in time, i.e., work must be finished on every member of the set before it can proceed to the machine where the next operation takes place. Most operations are *non-preemptive* but *breakable*, i.e., the workpieces cannot be unmounted from and re-mounted to the machines but after completing an operation. On the other hand, the on-going operations can be interrupted, e.g., during the weekends, and continued later on without any extra cost. However, some operations, such as heat treatment, must not be broken due to technological reasons. Some of the operations are preceded by a fixed-duration *setup* and *transportation*.

Each operation employs a single machine for a given length of time. During this time, no other operation can be executed on the same machine tool. Amongst these machines, there are machining and welding centers, assembly and inspection stations, and other highly specialized machines as well. Machines with identical capabilities are organized into homogeneous machine groups. Currently, there are ca. 80 unique machines and 10 more homogeneous groups of machines with a capacity of between 2 and 8. However, the number of machines is growing as the facility is extended continuously.

Operations also need the attendance of a worker from a given group. Currently, each of the ca. 200 individual workers belongs to one of the 10 different worker groups, like welders, NC machinists, or inspectors. We consider workers within the same group to be of identical skills. When a sensitive (e.g., welding) operation requires proficient workforce, the problem of selecting of the appropriate individual from the worker group is left to the shop foreman.

Both machine and worker capacities are time varying but fixed for the periods of the planning horizon in resource calendars. Machine availability is also influenced by maintenance. Periods of planned maintenance are given, and there is no need to do maintenance planning. When these so-called *normal* capacities do not suffice, it is possible to extend them by overtime or by renting workforce. These *extra* capacities are available for extra cost.

Tools, fixtures, storage and transportation means do not cause bottlenecks, hence, it is unnecessary to model them as finite capacity resources. In-process buffers are also unlimited. At the same time, some operations can only be executed on external resources, by subcontractors. These are the so-called *outsource processes* (OSP).

The factory is equipped with an enterprise resource planning (ERP) system that stores all order, technology, and resource related data, as well as a manufacturing execution system (MES) that collects on-line information about job shop status.

4.2 Problem Statement

Our goal within the project was to bridge the gap between the ERP and the MES systems of the factory. This required the development of an integrated *production planner and scheduler* (PPS) system with the following components.

- A *medium-term planner* whose challenge is the timing of production activities over a 3-6 months long time horizon.
- A *short-term scheduler* that is responsible for producing detailed, executable schedules that achieve the goals set by the production plans.
- A *simulator* whose role in the overall framework is to validate schedules and test their sensitivity towards uncertainties that are not included in the planning and scheduling models.

The medium-term planner should determine a temporal assignment of production activities so that project deadlines and raw material arrivals are observed. At the same time, it should respect the capacity constraints of the factory. When the involvement of extra capacities – overtime, subcontracting, etc. – is inevitable, it should minimize the *cost of extra capacity usage*. The plan should also keep the *work-in-process* (WIP) as low as possible. The planner will be run once a week to

prepare and update production plans for the next quarter or half of a year, with a rolling horizon.

The ultimate goal of the short-term scheduler is to unfold the medium-term plans into executable detailed schedules. The scheduler has to determine the order of the operations and the resource allocations with respect to all technological, temporal and capacity constraints. The optimization objective of the scheduler is to *minimize tardiness* with respect to the due dates set by the medium-term plan. Schedules will be prepared at the beginning of each week, but can be revised any time, when unexpected events occur.

Simulation captures such relevant aspects of the PPS problem that cannot be represented in the deterministic models of the planner and the scheduler. It validates the detailed schedules in the face of various uncertainties present at the shop floor. Moreover, the simulator took the role of the factory during the development period of the planner and scheduler system.

The planner, the scheduler and the simulator have to use the same master data readily available in the de facto standard production information systems of the factory. Furthermore, the solution methods applied at all levels have to be efficient enough to find close-to-optimal solutions quickly, supporting interactive decision making.

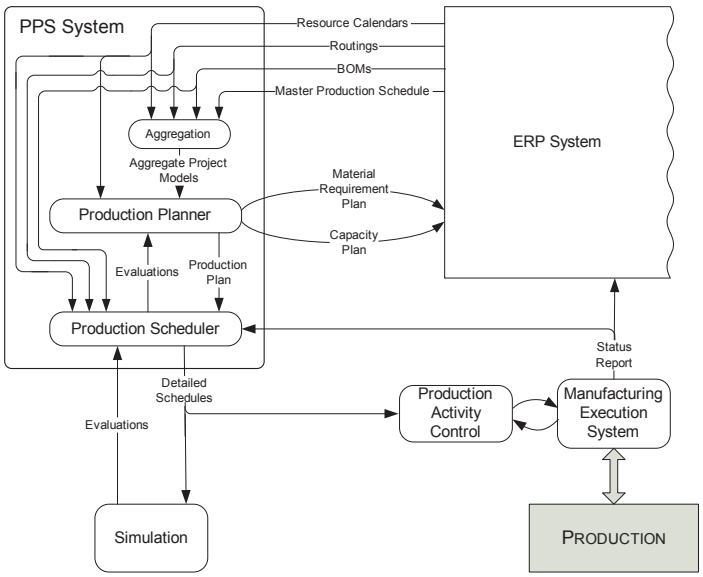


Figure 4.2: Architecture of the PPS system.

4.3 System Overview

The above requirements called for a hierarchical planner and scheduler architecture [60, 65], as shown in Fig. 4.2. The system was realized using Microsoft .NET component technology. The business logic components (the planner, the scheduler, and the aggregator) were encoded in C++. The planner and the scheduler were built on the top of a professional mathematical program solver [51] and a constraint solver [52] software, respectively. The simulation model was implemented by using an object-oriented discrete event simulation tool [86]. Proterv-II has a graphical user interface that facilitates its use as a decision support system at both levels. The user interface was encoded in Visual Basic.

The system stores all input, output and intermediate data in a Microsoft Access relational database. Fig. 4.3 gives a view of the tables in the database and their relations. Proterv-II communicates with the factory's database system via file interface. The sources of the input data are the following.

- BOMs, routings, the master production schedule and the resource calendars are available from the factory's enterprise resource planning (ERP) system.
- The maintenance management system (MMS) provides the actual machine availabilities with regard to planned maintenance.
- Shift-by-shift worker calendars can be obtained from a so-called time and attendance system (T&A).
- The factory's manufacturing execution system (MES) supplies our scheduler with on-line information about job shop status.

4.4 The Production Planner Sub-system

The challenge of the medium-term production planner sub-system is the timing of the production activities over a typically 3-6 months long time horizon with a time unit of one week. The generated plans must comply with the project deadlines and raw material arrivals, and respect the capacity constraints of the machines and workers of the factory. At the same time, they should keep production cost as low as possible by minimizing extra capacity usage and work-in-process. All in all, the production planner produces the following outputs.

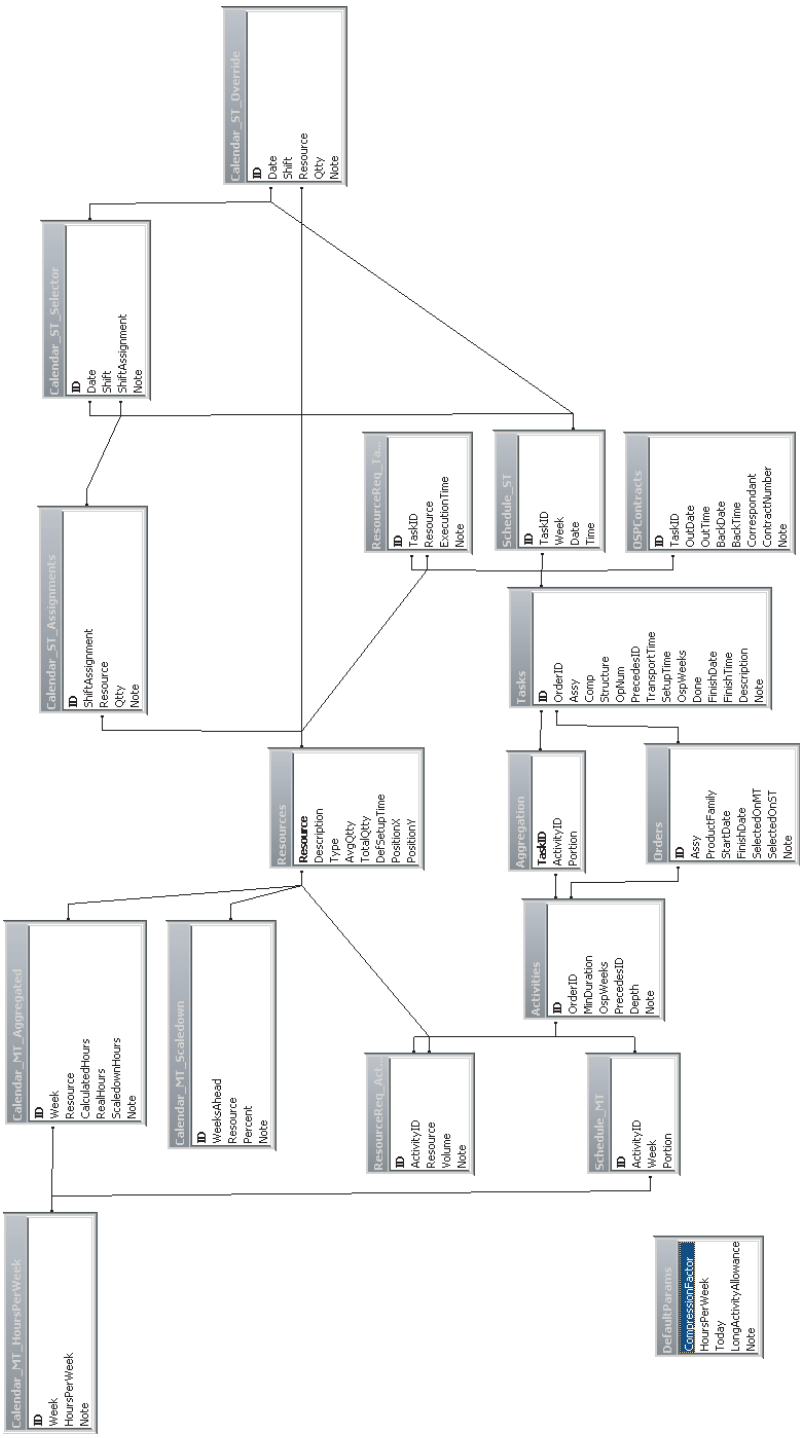


Figure 4.3: Tables in the database and their relations.

- A medium-term production plan that assigns operations to weeks of the planning horizon.
- A medium-term capacity plan that specifies the resource requirements of each week within the planning horizon.
- Suggested outgoing and incoming dates of OSP operations.

Furthermore, the medium-term plan can also serve as a material requirement plan that specifies the weekly requirements for raw materials and components. Clearly, this is practicable only for raw materials whose order period is calculable.

Due to the variance of products and the fluctuation of production load, all the above aims cannot be achieved by traditional material requirements planning (MRP) logic. Moreover, the planner must cope with a hard combinatorial optimization problem that cannot be solved for each individual manufacturing operation, because of the sheer size of the problem. Hence, we applied the aggregate production planning approach introduced in Chapter 2. In the production planning problem, we regard each customer order as one project. Aggregate models of projects are formed from the BOMs and routings by using the methods proposed in Sect. 2.3 and the bi-criteria tree partitioning algorithm described in Sect. 2.4.5. During the creation of the aggregate models, the throughput time of the activities is estimated by using the LFT heuristic (see Sect. 2.5.1). The result of the aggregation procedure is an in-tree of activities that are linked by end-to-start precedence relations. For example, from the routings displayed in Fig. 4.4 the system prepares the aggregate model with 5 activities presented in Fig. 4.5.[1]

Then, the production plan is prepared for the activities which *must* be performed within the planning horizon, based on their distance from the root activity in the project's aggregate model and the project's deadline. All the activities of a project have to fit into the project's time window set by the forecasted raw material arrival and the deadline.

We consider both machines and workers to be finite capacity resources. Activities may require different amounts of an arbitrary number of these resources for their processing. Normal weekly capacities of the resources can be computed from the resource calendars, by applying a *security factor* to the total working hours of the given resource. The security factor is indispensable because effective utilization of

[1] All data in the figures are distorted for reasons of confidentiality.

ID	Assy	Comp	Structure	OpNum	PrecedesID	TransportTime	Description
2774	872E0544G017	-	-	10	0	0.2	Packaging
2775	872E0544G017	SNP1734137	1	40	2774	0.2	FINAL INSPECTION
2776	872E0544G017	SNP1734137	1	30	2775	0.2	FINAL WELDING
2777	872E0544G017	SNP1734137	1	20	2776	0.2	AIR FLOW CONTROL
2778	872E0544G017	SNP1734137	1	10	2777	0.2	ASSEMBLY
2779	872E0544G017	SNR75886	1.6	80	2778	0.5	FINAL INSPECTION
2780	872E0544G017	SNR75886	1.6	70	2779	0.5	WELDING
2781	872E0544G017	SNR75886	1.6	60	2780	0.5	INSPECTION BEFORE WELDING
2782	872E0544G017	SNR75886	1.6	50	2781	0.5	Washing
2783	872E0544G017	SNR75886	1.6	40	2782	0.5	Deburring
2784	872E0544G017	SNR75886	1.6	30	2783	0.5	Turning
2785	872E0544G017	SNR75886	1.6	20	2784	0.5	Turning preparation
2786	872E0544G017	SNR75886	1.6	10	2785	0.5	Milling (OP1, OP2)
2787	872E0544G017	SNR75886	3.1.1	80	2774	0.2	FINAL INSPECTION
2788	872E0544G017	SNR75886	3.1.1	70	2787	0.2	WELDING
2789	872E0544G017	SNR75886	3.1.1	60	2788	0.2	INSPECTION BEFORE WELDING
2790	872E0544G017	SNR75886	3.1.1	50	2789	0.2	Washing
2791	872E0544G017	SNR75886	3.1.1	40	2790	0.2	Deburring
2792	872E0544G017	SNR75886	3.1.1	30	2791	0.2	Turning
2793	872E0544G017	SNR75886	3.1.1	20	2792	0.2	Turning preparation
2794	872E0544G017	SNR75886	3.1.1	10	2793	0.2	Milling (OP1, OP2)
2795	872E0544G017	SNR75886	3.26.1	80	2774	0.2	FINAL INSPECTION
2796	872E0544G017	SNR75886	3.26.1	70	2795	0.2	WELDING
2797	872E0544G017	SNR75886	3.26.1	60	2796	0.2	INSPECTION BEFORE WELDING
2798	872E0544G017	SNR75886	3.26.1	50	2797	0.2	Washing
2799	872E0544G017	SNR75886	3.26.1	40	2798	0.2	Deburring
2800	872E0544G017	SNR75886	3.26.1	30	2799	0.2	Turning
2801	872E0544G017	SNR75886	3.26.1	20	2800	0.2	Turning preparation
2802	872E0544G017	SNR75886	3.26.1	10	2801	0.2	Milling (OP1, OP2)
2803	872E0544G017	SNR75886	3.27.1	80	2774	0.2	FINAL INSPECTION
2804	872E0544G017	SNR75886	3.27.1	70	2803	0.2	WELDING
2805	872E0544G017	SNR75886	3.27.1	60	2804	0.2	INSPECTION BEFORE WELDING
2806	872E0544G017	SNR75886	3.27.1	50	2805	0.2	Washing
2807	872E0544G017	SNR75886	3.27.1	40	2806	0.2	Deburring
2808	872E0544G017	SNR75886	3.27.1	30	2807	0.2	Turning
2809	872E0544G017	SNR75886	3.27.1	20	2808	0.2	Turning preparation
2810	872E0544G017	SNR75886	3.27.1	10	2809	0.2	Milling (OP1, OP2)
2811	872E0544G017	SNR75886	3.28.1	80	2774	0.2	FINAL INSPECTION
2812	872E0544G017	SNR75886	3.28.1	70	2811	0.2	WELDING
2813	872E0544G017	SNR75886	3.28.1	60	2812	0.2	INSPECTION BEFORE WELDING
2814	872E0544G017	SNR75886	3.28.1	50	2813	0.2	Washing
2815	872E0544G017	SNR75886	3.28.1	40	2814	0.2	Deburring
2816	872E0544G017	SNR75886	3.28.1	30	2815	0.2	Turning
2817	872E0544G017	SNR75886	3.28.1	20	2816	0.2	Turning preparation
2818	872E0544G017	SNR75886	3.28.1	10	2817	0.2	Milling (OP1, OP2)
(AutoNumber)					0	0	

Figure 4.4: Routing of a product. Excerpt from the MS Access database.

ID	OrderID	MinDuration	OspWeeks	PrecedesID	Depth	Note
304	190	78.7	0	0	0	
305	190	72.2	0	304	1	
306	190	72.2	0	305	2	
307	190	90.4	0	304	1	
308	190	60.2	0	307	2	
(AutoNumber)	0	0	0	0	0	

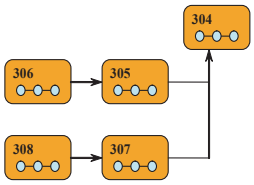


Figure 4.5: An aggregate project model. Excerpt from the MS Access database.

resources never reaches 100% in practice. By default, we set this factor to 0.8 for the upcoming weeks and 0.7 for distant weeks in order to allow for unforseen projects as well. Extra capacities are also limited, they extend the normal capacities to the

theoretical total working hours of the resource. However, there is a possibility to plan for infinite extra capacities as well.

In the aggregate project models, OSP operations take place in separate activities. It is assumed that the outgoing and incoming dates of the OSP operations fall on borders of the one-week time units. There are two ways to handle these dates. Those which are already negotiated and fixed in the input of the planner, post temporal constraints on the predecessors or successors of the corresponding OSP operations. Hence, the generated plan will always comply with the negotiated OSP dates. For other OSP operations, the dates specified in the plan can be regarded as a suggestion.

When solving the planning problem, our primary objective is to *minimize extra capacity usage*. In this way, the planner attempts to keep the works allotted for a medium-term horizon within the factory. There is also a secondary objective: in order to minimize inventory costs, the *WIP level* should be minimal.

The planning problem is translated into a mixed-integer linear program, and solved by the branch-and-cut algorithm described in Sect. 2.3. The proposed algorithm is *any-time*: it generates a series of solutions with better and better objective values, thus a first feasible solution is generated quickly, and then it can be refined to converge towards the optimum. The first phase of the solution process, i.e., finding a plan with minimum extra capacity usage generally takes a few seconds only. However, minimizing WIP subject to this capacity constraint requires a significantly higher effort. Finding a sufficiently good solution took between 5 and 60 minutes. At the same time, reaching the theoretically optimal WIP level was not always possible within a reasonable amount of time. Table 4.1 shows the typical size of the problems we tackled.

Fig. 4.6 presents a fragment of a production plan prepared by our PPS system. It is filtered for one of the product families. In this chart, the red and blue vertical lines delimit the planning horizon, and each row stands for one project. White bars show the allowed time windows of the projects, while green stripes indicate weeks when one or more activities of the given projects have to be performed. A low WIP level corresponds to relatively short green stripes shifted to the right hand side of the white time windows. This particular production plan reflects an overloaded factory where resource shortage causes temporary breaks in several projects.

Fig. 4.7 shows a capacity plan for the NC milling machinist worker group, for the same period of time. The height of the green columns in the figure indicates the total amount of work of the resource in the given weeks. The yellow and orange areas in

Planning horizon	15–30 weeks
Time unit	1 week
Running projects	600–1200
Operations per projects	20–500
Activities per projects	1–10
Resources, total	ca. 100
Individual machines	ca. 80
Machine groups	ca. 10
Worker groups	ca. 10
Solution time	
Minimal extra capacities	2–15 sec.
Minimal extra capacities, minimal WIP	$\geq$ 5 min.

Table 4.1: Typical size of a medium-term planning problem.

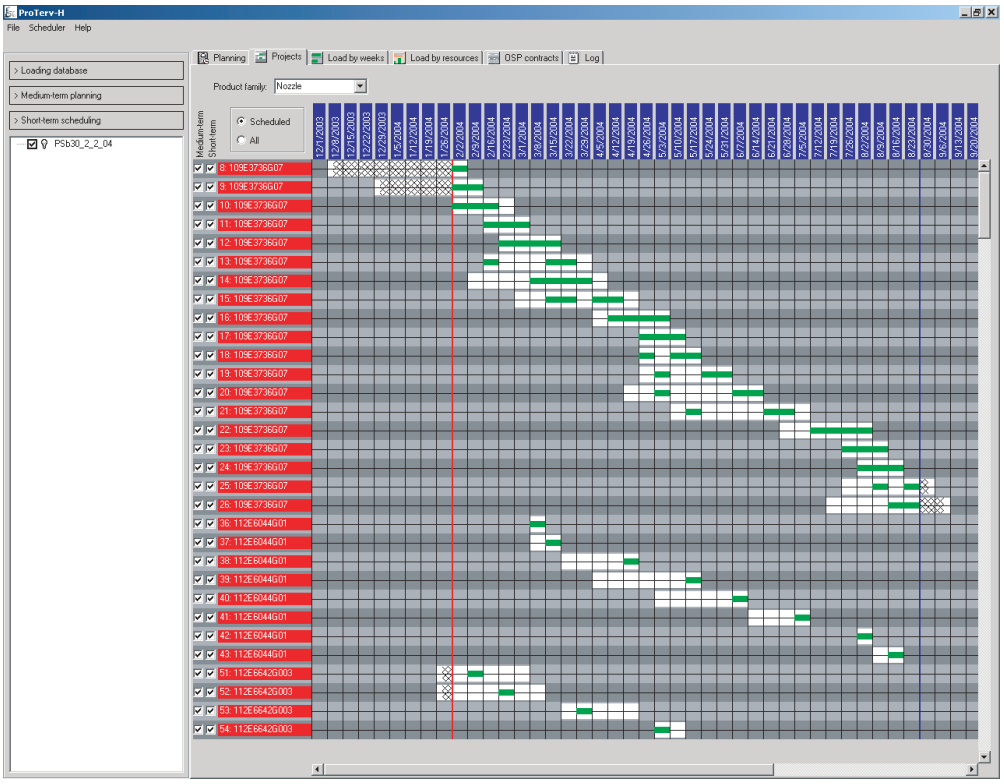


Figure 4.6: Project view of the production plan.

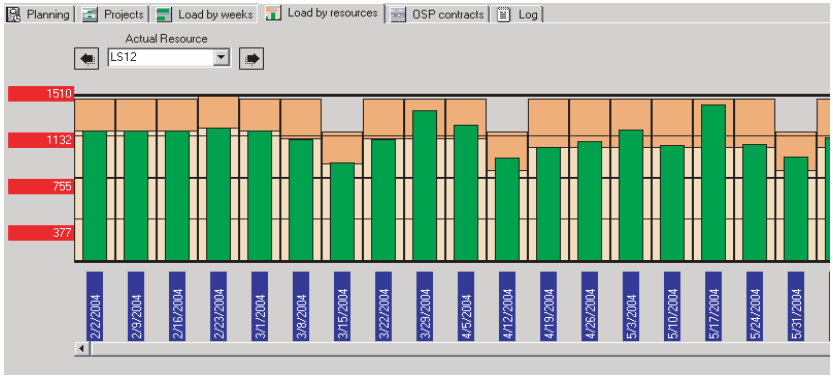


Figure 4.7: Resource view of the production plan.

the background show the limits of the normal and the extra capacities, respectively. Three capacity limits are lower than others because national holidays fall into the corresponding weeks. This capacity plan shows a contiguously overloaded group of workers: the normal capacities are completely exploited and in several weeks the use of overtime is also required.

The medium-term plan can be edited by the modification of the model's input parameters through the same user interface. The time windows of the projects can be adjusted by dragging the corresponding white bar in the production plan chart, and weekly resource capacities can be altered by using a similar technique, too. The check box which facilitates the usage of infinite extra capacities proved to be greatly helpful as well. It made possible the solution of some heavily overloaded test instances where the capacity requirements of projects could not be satisfied even by the extra capacities as described above.

4.5 The Production Scheduler Sub-system

In the integrated PPS framework, the responsibility of the scheduler is to achieve the goals set by the planner by unfolding its aggregate plans into realizable operation sequences. It has to determine the order of the operations on resources with respect to all technological, temporal and capacity constraints. Hence, the scheduler works with medium-term production plans, detailed resource calendars, as well as BOMs and routings (see also Fig. 4.2). In return, it generates detailed predictive production schedules that, besides satisfying the constraints, approach optimality with respect to the actual optimization criteria. The scheduling horizon is one week – i.e., equals

Scheduling horizon	1 week
Time unit	0.1 hour
Running projects	30–80
Operations per projects	20–200
Processing time of operations	0.5–120 hours
Resources, total	ca. 100
Machines (unary)	ca. 80
Machines (cumulative)	ca. 10
Workers (cumulative)	ca. 10
Solution time	$\geq$ 5 sec.

Table 4.2: Typical size of a short-term scheduling problem.

the time unit of the planner–, while the time unit of the scheduler is 0.1 hour.

Typically, schedules are generated for the next few weeks only. The work to be scheduled for a specific week is received by disaggregating the activities which fall into the given week in the medium-term plan. If the execution of an activity covers several weeks then its operations are distributed among the weeks proportional to the activity's intensities.

Each operation has its processing time, and an optional transportation and setup time. We assume that transportation and setup are performed before the operation, but while the former needs the workpiece only, setup employs solely the resources of the operation. Furthermore, operations may require several finite-capacity resources during their execution. The capacity of the resources can vary shift-by-shift. The typical size of our detailed production scheduling problems is presented in Table 4.2.

To be able to model all the above facets of the scheduling problem and solve it close-to-optimally, we took the constraint-based approach. An overview of the available methods for representing and solving scheduling problems by constraints has been given in Sect. 3.1. Below, we describe how we applied these techniques to solving practical scheduling problems in Proterv-II.

Although operations typically require one machine (e.g., a turning center) and one worker (a turning machinist) in the particular application, our model allows the operations to employ an arbitrary number of resources. Some longer operations do not occupy all the required resources – typically, the worker – during the whole length of their execution. In such cases we assume that all the resources start working at

the same time, but might finish individually, after the passing of a resource specific execution time. Hence, to an operation with n different execution times on different resources, n tasks correspond in the constraint-based representation. Each of the fixed duration tasks require the resources on which the execution of the operation equals the duration of the tasks. The start times of the tasks are bound together by an equality constraint, and the precedence constraints referring to the operation are posted on the longest task.

Transportation times can be modelled by delays attached to the precedence constraints between the corresponding operations. These generalized precedence constraints are propagated by the standard *arc-B-consistency* algorithm. There is a limited opportunity to represent *setup times* as well. From the scheduling aspect, we regard setup as the preliminary part of an operation which does not require the presence of the workpiece, but occupies one or more resources. Hence, the workpiece can undergo other manufacturing operations or transportation while the given machine is already busy with the adjustment to the forthcoming operation.[2]

Some of the resources, e.g., most of the machine tools are unique and able to process one operation at a time. These are modelled by *unary resources*. Homogeneous machine groups and worker groups are represented by cumulative resources. Their capacity varies shift-by-shift. In Proterv-II, both types of resource capacity constraints are propagated by the widely used *edge-finding* algorithm (see. Sect. 3.1.2).

As it has been stated, the objective of the scheduler is to achieve the goals set by the planner. Minimizing greatest delay w.r.t. the common due date of the weekly workload is equivalent to minimizing the makespan of the schedule – provided that there is a positive delay. Decreasing the makespan further is favorable even when there is no positive delay, because – roughly speaking – it increases the parallelism between the projects and improves the resistance of the schedule against disturbances. All in all, we applied *makespan* as the optimization criterion at the scheduling level.

For the solution of the resulting constraint program, we applied a *branch-and-*

[2]The applied search strategy with a *setting times* branching scheme (see Sect. 3.1.3) is complete and efficient only if

$$\forall t_1, t_2 \in T : (t_1 \rightarrow t_2) \Rightarrow s_{t_1} \leq s_{t_2} + d_{t_2} + u_{t_1,t_2}$$

holds, where s_t denotes the setup time associated with task t and u_{t_1,t_2} stands for the time of transportation between t_1 and t_2. This condition ensures that the chronological search strategy will consider each task only after its predecessors' start times have been bound. Consequently, we *truncate* longer setup times to meet this criterion. In fact, more sophisticated methods exist in the scheduling literature to handle setups [1], but those are computationally more complex and the enterprise of our application could not provide them with the required detailed data.

bound search using a *setting times* branching strategy (see Sect. 3.1.3). This solution method is – like that of the medium-term planner – *any-time*, which enables our scheduler to be used interactively. Since the novel algorithms proposed in Sect. 3.2 were elaborated after the development of Proterv-II, they do not constitute an integrated part of this scheduler.

Proterv-II allows the user to prepare detailed schedules for selected weeks of the medium-term horizon. It presents the generated schedules in project- and resource-oriented Gantt charts. The charts can be filtered for individual projects or resources, and displayed with daily, shift-by-shift or hourly granularity. The project view of the Gantt chart shows the operations of the activities distributed in the week, as presented in Fig. 4.8. Green rectangles stand for the operations, while the black arrows connecting them represent precedence constraints. Each row of the figure corresponds to the production of one component within a project. The resource view of the Gantt chart shows the order of operations to be executed by a particular resource. Fig. 4.9 presents the Gantt chart of the fitter worker group. The current implementation of Proterv-II does not provide any functionality to edit the short-term schedules manually.

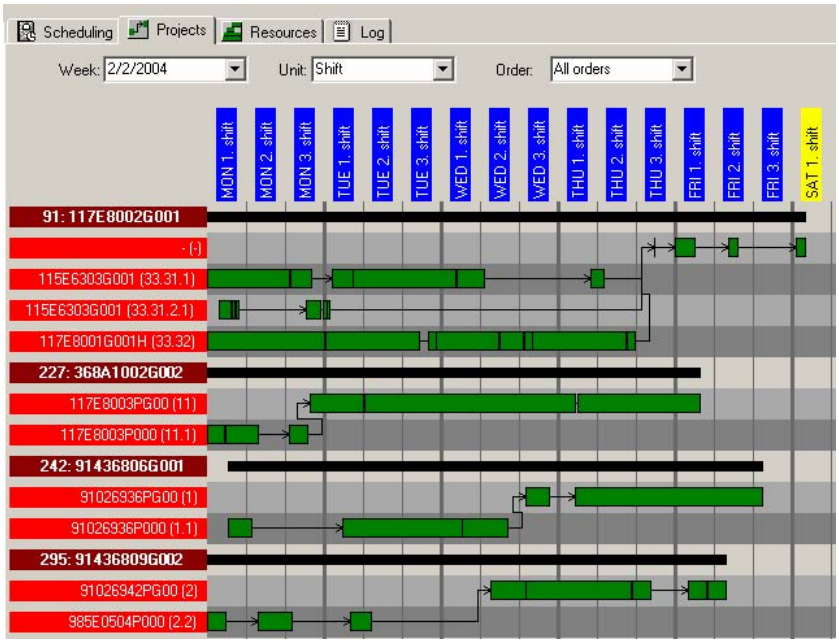


Figure 4.8: Project view of a detailed schedule.

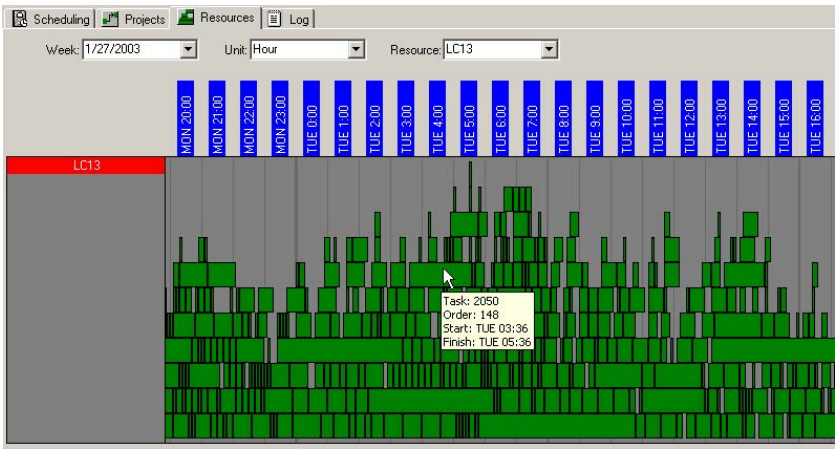


Figure 4.9: Resource view of a detailed schedule.

4.6 Verification of the Results by Simulation

The planning and scheduling models considered so far in this thesis were deterministic, and disregarded all types of uncertainty. However, the real-life production environment in which the plans and schedules are to be executed is inevitably perturbed by various types of unexpected events. Sources of uncertainty in this factory include the following.

- Downtimes, i.e., machine failures and unexpected absence of workforce.
- Processing, setup and transportation times that depend on the proficiency of the worker or on other random factors.
- Adjustment operations that may have to be inserted into the schedules depending on the results of quality checks.
- Erratic raw material arrival.

Hence, the ability to evaluate the behavior of the plans and schedules in the face of uncertainties is of key importance. To that end, our colleagues applied discrete event simulation. The interface of the simulator can be seen in Fig. 4.10. Authors of [55] analyzed situations where processing times could vary between the 90 and 130 percent of the planned value, machines and workers were unavailable at the 10 or 20 percent of their planned working hours, and the efficiency of new employees was decreased by 5 to 25 percents. During the simulation runs, the start times of operations in the

detailed schedules were relaxed and only the order of the operations on each resource was kept. Delay caused by the disorders w.r.t. the medium-term plan was measured in the number of late tasks and the average and maximum tardiness.

The simulation experiments have shown that in most cases the medium-term plans are robust enough to remain feasible despite the unexpected events. Operations which could not be executed at the proper week were admitted by the schedule of the next week, mostly without violating any customer deadlines. Using the same simulation model, the authors could perform a sensitivity analysis, too. For example, they pointed out the worker groups in which unexpected absence or decreased efficiency can cause considerable lateness. A more detailed description of the simulation study is out of the scope of this thesis. In this topic, readers should refer to [55].

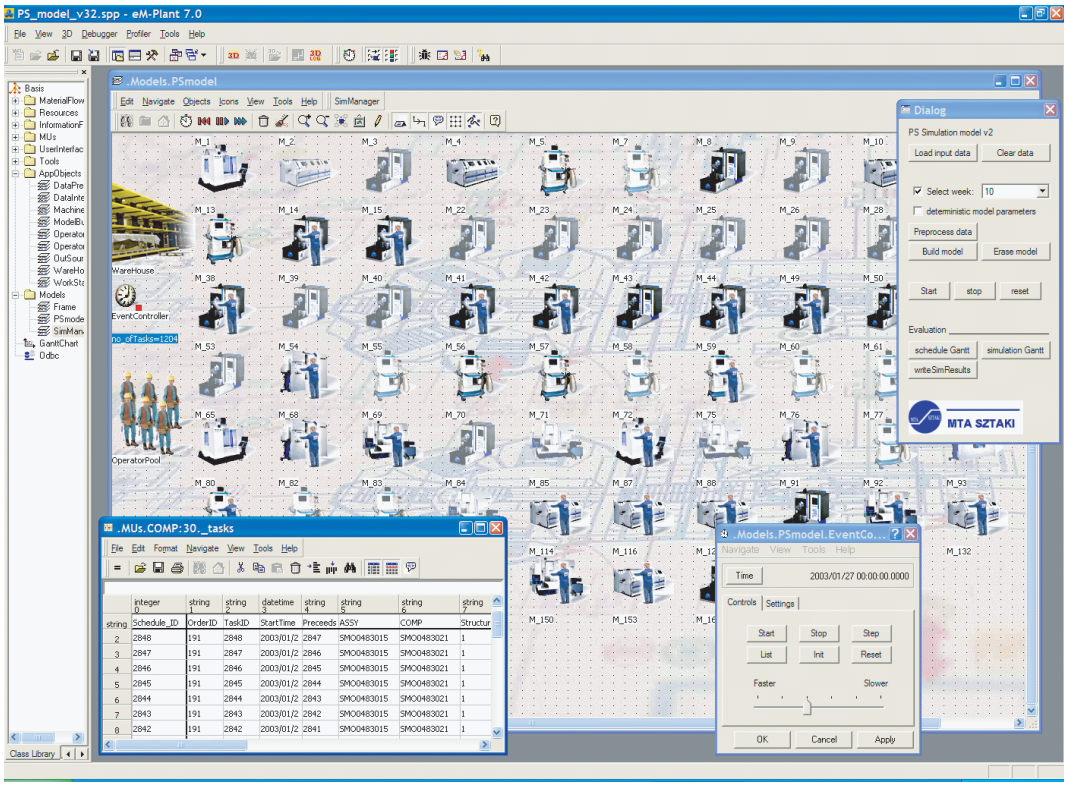


Figure 4.10: The interface of the simulator [55].

Chapter 5

Conclusions

In this thesis, we investigated production planning and scheduling in make-to-order production systems. We argued that these problems need clear-cut models that capture the relevant aspects of production, and efficient algorithms to find their close-to-optimal solutions in a reasonable time.

For medium-term production planning, this required us to define a novel formulation of the aggregate production planning problem. We proposed fast, polynomial-time tree partitioning algorithms for constructing the best suited problem representation from detailed technology, order, and capacity related data. To our knowledge, this approach is the first to seek the feasibility of the production plan by linking the production planning model to the detailed scheduling representation. This link is of crucial practical significance, since it makes possible the automated construction of the production planning problem from data readily available in de facto standard enterprise information systems, without the involvement of human experts. Experiments performed on industrial test problems confirmed that our approach – with appropriate extensions – can capture the relevant aspects of production planning, and leads to production plans that can be unfolded into executable detailed schedules.

For detailed scheduling, we defined two novel consistency preserving transformations for the solution of constraint-based scheduling problems. We argued that extending current constraint solvers by such transformations can boost their performance on typical, structured problem instances originating from the industry. Specifically, *progressive schedules* exploited the presence of many similar projects in the factory. This transformation is profitable in any computational paradigm where a search space reduction can be converted into computational efficiency. In contrast,

freely completable partial solutions reveal components of a problem which are relatively easy to solve and are only loosely connected to the remainder of the problem. They help eliminate the irrelevant decisions from the search tree. In our experiments, this resulted in a two orders of magnitude decrease of the search tree size. These results have shown that practical problem instances often contain a hidden structure that, if exploited by appropriate means, can provide the clue to the efficient solution of even the most complex combinatorial optimization problems.

Beyond presenting our theoretical results, we reported on the development of Proterv-II, a pilot integrated production planner and scheduler software. This system served as the test bed of the models and algorithms proposed in this thesis, in experiments run on real-life planning and scheduling problems. We believe that the achieved results enable an extended version of Proterv-II to proceed towards a true industrial application.

Acknowledgements[1]

Although there is only a single name displayed on the front page as the author of this thesis, its content is the fruit of a four years teamwork with my advisors, colleagues – and also friends. Most contained ideas emerged during our abundant discussions, and even the current presentation passed the hands of many people. Below, I would like to thank the ones who helped the most.

At the first place, I am indebted to my advisors, Dr. József Váncza and Dr. Tadeusz Dobrowiecki. I learnt a lot from them during the years spent together, and not only about scheduling or artificial intelligence. I benefited much from proficiency of Dr. Tamás Kis during our numerous conversations. I am grateful to Péter Egri for taking a great part in the implementation of the Proterv-II system. And I also owe one to Dr. Gábor Erdős, who helped me finish this work in several unorthodox ways, e.g., by asking me three times a day whether my thesis is complete. I thank Zoltán Mihályi, Attila Szántai, and János Gyapalyi from GE Hungary for providing the industrial background of our research and supplying us with data that enabled the validation of our results. I thank Dr. Sigrid Knust for sharing with us their tabu search based RCPSP solver.

I enjoyed working together with the members of the Laboratory of Engineering and Management Intelligence, headed by Prof. László Monostori. I also took pleasure in my teaching practice spent at the Department of Measurement and Information Systems, lead by Prof. Gábor Péceli. And finally, the inspiration that I was provided by Prof. András Márkus is unforgettable, even if I was not provided to have his control on the final outcome of my PhD studies.

[1]This work has been supported by NRDP grants No. 2/040/2001 and 2/010/2004, OTKA grant No. T046509, and the VRL-KCiP NMP2-CT-2004-507487 and DECOS EU FP 6 projects.

List of Abbreviations

BOM	Bill Of materials
DAG	Directed acyclic graph
ERP	Enterprise resource planning
FCPS	Freely completable partial solution
LFT	Latest finish time / a priority rule that sorts tasks by their increasing lft
LFT^{rand}	Randomized version of the LFT rule
LP	Linear programming
MES	Manufacturing execution system
MILP	Mixed-integer linear programming
MMS	Maintenance management system
MRP	Material requirements planning
MRP II.	Manufacturing resources planning
NC	Numerical Control
OPL	The Optimization Programming Language
OSP	Outsource process
PPS	Production planning and scheduling
RCPSP	Resource-constrained project scheduling problem
RCPSVP	Resource-constrained project scheduling problem with variable-intensity activities
SAT	Boolean satisfiability problem
SBDD	Symmetry Breaking via Dominance Detection
SBDS	Symmetry Breaking During Search
T&A	Time and attendance system
WIP	Work-in-process

Notations

Aggregate Production Planning

Γ^*	Optimal detailed schedule
Γ_Π	Detailed schedule, obtained by disaggregating the aggregate plan Π
Θ	Length of the aggregate time unit
μ_A	Activity security factor
μ_R	Resource security factor
Π^*	Optimal aggregate plan
ϱ_r^A	Amount of work required by activity A on resource r
τ	Time unit
A	Activity
$\mathbf{A}$	Overall set of activities
c_τ^r	Cost of extra capacity usage for each unit of resource r
$d(A)$	Minimum throughput time of activity A
d_t	Duration of task t
end_t	End time of task t
est_A	Earliest start time of activity A
est_P	Earliest start time of project P
j_A	Maximal intensity of activity A
lft_A	Latest finish time of activity A
lft_P	Latest finish time of project P
P	Project
$\mathbf{P}$	Overall set of projects
$q(r)$	Capacity of resource r (detailed scheduling)
q_τ^r	Normal capacity of resource r at time τ (aggregate planning)
$\hat{q}_\tau^r$	Extra capacity of resource r at time τ (aggregate planning)
$\mathbf{R}$	Overall set of resources
$r(t)$	Resource required by task t
$start_t$	Start time of task t
$start_P$	Start time of project P
t	Task
T	Overall set of tasks

T_P	Tasks of project P
$WIP(\Gamma)$	Work-in-process value of the schedule Γ
w_P	Weight factor of project P
x_τ^A	Intensity of activity A at time τ

Constraint Programming, Constraint-based Scheduling

β	Bijection between two pairs of tasks
Π	Constraint program
τ	Time unit
c	Constraint
C	Overall set of constraints in a constraint program
D	Set of variable domains in a constraint program
D_i	Domain of variable x_i
d_t	Duration of task t
$Dmin(x)$	Minimum of the domain of variable x
$Dmax(x)$	Maximum of the domain of variable x
eft_t	Earliest finish time of task t
end_t	End time of task t
end_t^S	End time of task t in schedule S
est_t	Earliest start time of task t
LB	Lower bound
lft_t	Latest finish time of task t
lst_t	Latest start time of task t
$M_{r,\tau}^+$	Set of tasks that are under execution at time τ on resource r
$M_{r,\tau}^-$	Set of tasks that might be under execution at time τ on resource r
O	Objective function
p	Task
P	Set of tasks
PS	Partial solution
q	Task
Q	Set of tasks
$r(t)$	Resource required by task t

S	Solution of a constraint program (a schedule in case of scheduling problems)
s_t	Setup time associated with task t
$start_t$	Start time of task t
$start_t^S$	Start time of task t in schedule S
$q(r)$	Capacity of resource r
T	Overall set of tasks
$T(r)$	Set of tasks processed on resource r
T_τ	Set of tasks that can be processed at time τ (LFT heuristic)
T^{PS}	Set of tasks in partial solution PS
U	Set of tasks
UB	Upper bound
u_{t_1,t_2}	Transportation time between tasks t_1 and t_2
v_i^S	Value of variable x_i in solution S
X	Overall set of variables in a constraint program
X_c	Set of variables present in constraint c
x_i	Variable in a constraint program
X^{PS}	Set of variables in partial solution PS

Tree Partitioning

$\varphi(v)$	Partitioning option of sub-tree $T(v)$
$\Phi(v)$	Set of partitioning options of sub-tree $T(v)$
$\Psi_{k,q}$	Selection of partitioning options in the dynamic program
E	Edge set of a tree
h	Constant (denotes height)
$h(P)$	Height of partitioning P
$h(T)$	Height of tree T
$h_{min}(T)$	Minimal height of the admissible partitionings of T
K	Set of vertices
$l_{P_v}(u)$	Level of a vertex u with respect to a partitioning P_v
P	Partitioning
P_v	Partitioning of sub-tree $T(v)$
P_v^*	Optimal partitioning of sub-tree $T(v)$

$PO(v)$	Set of Pareto optimal partitionings of $T(v)$
$PO^h(v)$	Set of Pareto optimal partitionings of $T(v)$ with height of at most h
q	Constant (denotes cardinality)
$q(P)$	Cardinality of partitioning P
$q_{min}(T)$	Minimal cardinality of the admissible partitionings of T
Q_v	Partitioning of sub-tree $T(v)$
r	Root
$RC(P)$	Root component of partitioning P
$rw(P)$	Root component weight of partitioning P
$S(v)$	Set of the sons of vertex v
ST	Sub-tree
T	Tree
T^P	Tree obtained by contracting each sub-tree $ST \in P$ into a vertex
$T(v)$	Sub-tree rooted at vertex v
v	Vertex
u	Vertex
V	Vertex set of a tree
$V(ST)$	Vertex set of sub-tree ST
w	Weight function
W	Weight limit

Index

Bibliography

[1] A. Allahverdi, J. Gupta, and T. Aldowaisan. A review of scheduling research involving setup considerations. *Omega*, 27(2):219–239, 1999.

[2] K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[3] S. Axsäter. On the feasibility of aggregate production plans. *Operations Research*, 34(5):796–800, 1986.

[4] H. Aytug, K. Kempf, and R. Uzsoy. Measures of subproblem criticality in decomposition algorithms for shop scheduling. *International Journal of Production Research*, 41(5):865–882, 2003.

[5] T. Baar, P. Brucker, and S. Knust. Tabu-search algorithms and lower bounds for the resource-constrained project scheduling problem. In S. Voss, S. Martello, I. Osman, and C. Roucairol, editors, *Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 1–18. Kluwer, 1998.

[6] R. Backofen and S. Will. Excluding symmetries in constraint-based search. *Constraints*, 7(3):333–349, 2002.

[7] Ph. Baptiste and C. Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, 5(1/2):119–139, 2000.

[8] Ph. Baptiste, C. Le Pape, and W. Nuijten. Constraint-based optimization and approximation for job-shop scheduling. In *Proc. of AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems (IJCAI-95)*, pages 5–16, 1995.

[9] Ph. Baptiste, C. Le Pape, and W. Nuijten. Satisfiability tests and time-bound adjustments for cumulative scheduling problems. *Annals of Operations Research*, 92:305–333, 1999.

[10] Ph. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based Scheduling*. Kluwer Academic Publishers, 2001.

[11] Ph. Baptiste, L. Peridy, and E. Pinson. A branch and bound to minimize the number of late jobs on a single machine with release time constraints. *European Journal of Operational Research*, 144(1):1–11, 2003.

[12] J.E. Beasley. The OR-Library (Visited June 1, 2005), `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`.

[13] J.Ch. Beck, A.J. Davenport, and M.S. Fox. Five pitfalls of empirical scheduling research. In *Proc. of the 3rd International Conference on Principles and Practice of Constraint Programming (Springer LNCS 1330)*, pages 390–404, 1997.

[14] J.Ch. Beck and M.S. Fox. Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. *Artificial Intelligence*, 117:31–81, 2000.

[15] R.I. Becker and Y. Perl. The shifting algorithm technique for the partitioning of trees. *Discrete Applied Mathematics*, 62:15–34, 1995.

[16] G.R. Bitran, E.A. Haas, and A.C. Hax. Hierarchical production planning: a two-stage system. *Operations Research*, 30(2):232–251, 1982.

[17] J. Błażewicz, W. Domschke, and E. Pesch. The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research*, 93(1):1–33, 1996.

[18] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[19] P. Brucker. Complex scheduling problems. *Osnabrücker Schriften zur Mathematik*, P214 (Preprints), 1999. Available from `http://citeseer.ist.psu.edu/brucker99complex.html`.

[20] P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3–41.

[21] J. Carlier and E. Pinson. A practical use of Jackson's pre-emptive schedule for solving the job-shop problem. *Annals of Operations Research*, 26:269–287, 1990.

[22] J. Carlier and E. Pinson. Adjustments of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.

[23] A. Cesta, A. Oddi, and S.F. Smith. Profile-based algorithms to solve multiple capacitated metric scheduling problems. In *Proc. of the 4th International Conference on Artificial Intelligence Planning Systems*, pages 214–223, 1998.

[24] J. Crawford, G. Luks, M. Ginsberg, and A. Roy. Symmetry breaking predicates for search problems. In *Proc. of the 5th International Conference on Knowledge Representation and Reasoning*, pages 148–159, 1996.

[25] S. Dauzere-Peres and J.B. Lasserre. *An Integrated Approach in Production Planning and Scheduling*. Number 411 in Lectures Notes in Economics and Mathematical Systems. Springer-Verlag, 1994.

[26] E. Davis and J. Patterson. A comparision of heuristic and optimum solutions in resource-constrained project scheduling. *Management Science*, 21:944–955, 1975.

[27] E.L. Demeulemeester and W.S. Herroelen. A branch-and-bound procedure for the multiple resource-constrained project scheduling problem. *Management Science*, 38(12):1803–1818, 1992.

[28] E.L. Demeulemeester and W.S. Herroelen. *Project Scheduling: A Research Handbook*. Kluwer Academic Publishers, 2002.

[29] J. Erschler. *Analyse sous contraintes et aide à la décision pour certains problèmes d'ordonnancement*. PhD thesis, Université Paul Sabatier, 1976.

[30] J. Erschler, G. Fontan, and C. Merce. Consistency of the disaggregation process in hierarchical planning. *Operations Research*, 34(3):464–469, 1986.

[31] J. Erschler, P. Lopez, and C. Thuriot. Raisonnement temporel sous contraintes de ressources et problèmes d'ordonnancement. *Revue d'Intelligence Artificielle*, 5(3):7–32, 1991.

[32] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *Proc. of the 7th International Conference on Principles and Practice of Constraint Programming (Springer LNCS 2239)*, pages 93–107, 2001.

[33] B. Fleischmann and H. Meyr. Planning hierarchy, modeling and advanced planning systems. In A.G. de Kok and S.C. Graves, editors, *Supply Chain Management: Design, Coordination and Operation*, volume 11 of *Handbooks in Operations Research and Management Science*, pages 457–523. Elsevier, 2003.

[34] P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *Proc. of the 8th International Conference on Principles and Practice of Constraint Programming (Springer LNCS 2470)*, pages 462–476, 2002.

[35] F. Focacci, F. Laburthe, and A. Lodi. Local search and constraint programming. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 369–403. Kluwer Academic Publishers, 2003.

[36] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proc. of the 7th International Conference on Principles and Practice of Constraint Programming (Springer LNCS 2239)*, pages 77–92, 2001.

[37] M.S. Fox, S. Smith, B. Allen, G. Strohm, and F.C. Wimberly. ISIS: A constraint-directed reasoning approach to job-shop scheduling. In *Proc. of the IEEE Trends and Applications Conference*, pages 76–81, 1983.

[38] M. Frank. The Braess paradox. *Mathematical Programming*, 20(3):283–302, 1981.

[39] M.R. Garey and D.S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-completeness*. Freeman, 1979.

[40] I.P. Gent, W. Harvey, and T. Kelsey. Groups and constraints: Symmetry breaking during search. In *Proc. of the 8th International Conference on Principles and Practice of Constraint Programming (Springer LNCS 2470)*, pages 415–430, 2002.

[41] I.P. Gent, W. Harvey, T. Kelsey, and S. Linton. Generic SBDD using computational group theory. In *Proc. of the 9th International Conference on Principles and Practice of Constraint Programming (Springer LNCS 2833)*, pages 333–347, 2003.

[42] I.P. Gent and B.M. Smith. Symmetry breaking in constraint programming. In *Proc. of the 14th European Conference on Artificial Intelligence*, pages 599–603, 2000.

[43] S.T. Hackman and R.C. Leachman. An aggregate model of project-oriented production. *IEEE Transactions on Systems, Man and Cybernetics*, 19(2):220–231, 1989.

[44] A. Hallefjord and S. Storøy. Aggregation and disaggregation in integer programming problems. *Operations Research*, 38(4):619–623, 1990.

[45] A. Hamacher, W. Hochstättler, and C. Moll. Tree partitioning under constraints – clustering for vehicle routing problems. *Discrete Applied Mathematics*, 99:55–69, 2000.

[46] E.W. Hans. *Resource Loading by Branch-and-Price Techniques*. PhD thesis, Twente University, 2001.

[47] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[48] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *Proc. of IJCAI'95 – the 14th International Joint Conference on Artificial Intelligence*, pages 607–613, 1995.

[49] C. Holt, F. Modiglinai, and H.A. Simon. A linear decision rule for production and employment scheduling. *Management Science*, 2(1):1–30, 1955.

[50] H.H. Holtsclaw and R. Uzsoy. Machine criticality measures and subproblem solution procedures in shifting bottleneck methods: A computational study. *Journal of the Operational Research Society*, 47(5):666–677, 1996.

[51] Ilog. *Ilog Cplex 7.1 User's Manual*, 2001.

[52] Ilog. *Ilog Scheduler 5.1 User's Manual*, 2001.

[53] D.S. Johnson and K.A. Niemi. On knapsacks, partitions, and a new dynamic programming technique for trees. *Mathematics of Operations Reserach*, 8(1):1–14, 1983.

[54] N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, 2002.

[55] B. Kádár, A. Pfeiffer, and L. Monostori. Discrete event simulation for supporting production planning and scheduling decisions in Digital Factories. In *Proc. of the 37th CIRP International Seminar on Manufacturing Systems*, pages 441–448, 2004.

[56] T. Kis. A branch-and-cut algorithm for scheduling of projects with variable-intensity activities. *Mathematical Programming, in print*, 2005.

[57] R. Kolisch and S. Hartmann. Heuristic algorithms for the resource-constrained project scheduling problem: Classification and computational analysis. In J. Weglarz, editor, *Project Scheduling: Recent Models, Algorithms and Applications*, pages 147–178. Kluwer, 1999.

[58] A. Kovács. A novel approach to aggregate scheduling in project-oriented manufacturing. In *Proc. of the 13th International Conference on Automated Planning and Scheduling, Doctoral Consortium*, pages 63–67, 2003.

[59] A. Kovács. A novel approach to aggregate scheduling in project-oriented manufacturing. *Projects & Profits*, pages 73–80, October 2004.

[60] A. Kovács, P. Egri, and J. Váncza. Integrált termeléstervezés és ütemezés megrendelésre történő gyártásban (Integrated production planning and scheduling in make-to-order manufacturing. In Hungarian). *Gépgyártás, submitted*, 2004.

[61] A. Kovács and T. Kis. Partitioning of trees for minimizing height and cardinality. *Information Processing Letters*, 89(4):181–185, 2004.

[62] A. Kovács and J. Váncza. Constraint feedback in solving incomplete models: A case study in sheet metal bending. In *STAIRS 2002 – Proc. of the Starting Artificial Researchers Symposium*, pages 109–118, 2002.

[63] A. Kovács and J. Váncza. Completable partial solutions in constraint programming and constraint-based scheduling. In *Proc. of the 10th International Conference on Principles and Practice of Constraint Programming (Springer LNCS 3258)*, pages 332–346, 2004.

[64] A. Kovács, J. Váncza, and A. Márkus. Structural exploration of constraint-based scheduling problems. In *Proc. of the 37th CIRP International Seminar on Manufacturing Systems*, pages 433–439, 2004.

[65] A. Kovács, J. Váncza, L. Monostori, B. Kádár, and A. Pfeiffer. Real-life scheduling using constraint programming and simulation. In *Intelligent Manufacturing Systems 2003 – Proc. of the 7th IFAC Workshop on Intelligent Manufacturing Systems*, pages 213–218, 2003.

[66] S. Kundu and J. Misra. A linear tree partitioning algorithm. *SIAM Journal on Computing*, 6:151–154, 1977.

[67] Ph. Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: existing approaches and new results. *Artificial Intelligence*, 143(2):151–188, 2003.

[68] Ph. Laborie and M. Ghallab. Planning with shareable resource constraints. In *Proc. of IJCAI'95 – the 14th International Joint Conference on Artificial Intelligence*, pages 1643–1649, 1995.

[69] R.C. Leachman, A. Dincerler, and S. Kim. Resource constrained scheduling of projects with variable intensity activities. *IIE Transactions*, 22(1):31–40, 1990.

[70] R.C. Leachman and S. Kim. A revised critical path method for networks including both overlap relationships and variable-duration activities. *European Journal of Operational Research*, 64:229–248, 1993.

[71] R. Leisten. An LP-aggregation view on aggregation in multi-level production planning. *Annals of Operations Research*, 82(1):229–248, 1998.

[72] O. Lhomme. Consistency techniques for numeric CSPs. In *Proc. of IJCAI'93 – the 13th International Joint Conference on Artificial Intelligence*, pages 232–238, 1993.

[73] M. Maravalle, B. Simeone, and R. Naldini. Clustering on trees. *Computational Statistics & Data Analysis*, 24(2):217–234, 1997.

[74] A. Márkus, J. Váncza, T. Kis, and A. Kovács. Project scheduling approach to production planning. *CIRP Annals – Manufacturing Technology*, 52(1):359–362, 2003.

[75] A. Márkus, J. Váncza, and A. Kovács. Constraint-based process planning in sheet metal bending. *CIRP Annals – Manufacturing Technology*, 51(1):425–428, 2002.

[76] J.G. Monks. *Operations Management – Theory and Problems*. McGraw-Hill, 1987.

[77] T.E. Morton and D.W. Pentico. *Heuristic Scheduling Systems: With Applications to Production Systems and Project Management*. Wiley Series in Engineering and Technology Management. Wiley & Sons, 1993.

[78] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. Wiley & Sons, 1988.

[79] Oracle. *Oracle Advanced Scheduler*, 2003. Available from `http://www.oracle.com/applications/service/adv_sched_datasheet.pdf`.

[80] G. Pesant and M. Gendreau. A view of local search in constraint programming. In *Proc. of the 2nd International Conference on Principles and Practice of Constraint Programming (Springer LNCS 1118)*, pages 353–366, 1996.

[81] S. Prestwich. Combining the scalability of local search with the pruning techniques of systematic search. *Annals of Operations Research*, 115:51–72, 2002.

[82] I. Razgon and A. Meisels. Maintaining dominance consistency. In *Proc. of the 9th International Conference on Principles and Practice of Constraint Programming (Springer LNCS 2833)*, pages 945–950, 2003.

[83] D.F. Rogers, R.D. Plante, R.T. Wong, and J.R. Evans. Aggregation and disaggregation techniques and methodology in optimization. *Operations Research*, 39(4):553–582, 1991.

[84] SAP. *Production Planning and Detailed Scheduling with SAP Advanced Planner & Optimizer*, 2002. Available from `http://www.sap.com/solutions/business-suite/scm/brochures`.

[85] J. Slaney and T. Walsh. Backbones in optimization and approximation. In *Proc. of IJCAI-01 – the 17th International Joint Conference on Artificial Intelligence*, pages 254–259, 2001.

[86] Technomatix. *eM-Plant 5.5 Reference Manual*, 2000.

[87] E. Toczyłowski and K. Pieńkosz. Restrictive aggregation of items in multi-stage production systems. *Operations Research Letters*, 10(3):159–163, 1991.

[88] P. Tormos and A. Lova. Tools for resource-constrained project scheduling and control: forward and backward slack analysis. *Journal of the Operational Research Society*, 52(7):779–788, 2001.

[89] P. Torres and P. Lopez. On not-first/not-last conditions in disjunctive scheduling. *European Journal of Operational Research*, 127:332–343, 2000.

[90] P. Torres and P. Lopez. Overview and possible extensions of shaving techniques for job-shop problems. In *Proc. of the 2nd International Workshop on Integration of AI and OR techniques (CP-AI-OR'2000)*, pages 181–186, 2000.

[91] T. Tóth, F. Erdélyi, and S. Radeleczki. Similarity-based project planning in the field of the production of individual machines. In *Proc. of the 37th CIRP International Seminar on Manufacturing Systems*, pages 225–230, 2004.

[92] M. Urgo. Personal communication, 2005.

[93] S. Vajda. *Mathematical Programming*. Addison-Wesley, 1961.

[94] P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

[95] J. Váncza, T. Kis, and A. Kovács. Aggregation – the key to integrating production planning and scheduling. *CIRP Annals – Manufacturing Technology*, 53(1):377–380, 2004.

[96] T. E. Vollmann, Berry W.L., and Whybark D.C. *Manufacturing Planning and Control Systems*. McGraw-Hill, 1997.

[97] M.G. Wallace. Practical applications of constraint programming. *Constraints*, 1(1):139–168, 1996.

[98] R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proc. of IJCAI'03 – the 18th International Joint Conference on Artificial Intelligence*, pages 1173–1178, 2003.

[99] A. Wolf. Better propagation for non-preemptive single-resource constraint problems. In *Proc. of the CSCLP04 – Joint Annual Workshop of ERCIM / CoLogNet on Constraint Solving and Constraint Logic Programming*, pages 230–243, 2004.