Scheduling of Tests on Vehicle Prototypes

Doctoral Thesis
Kamol Limtanyakul

Dissertation zur
Erlangung des akademischen Grades eines Doktors an der
Fakultät für Elektrotechnik und Informationstechnik
Technische Universität Dortmund

Abteilung Informationstechnik
Institut für Roboterforschung

September 30, 2009

Acknowledgements

I was told that to do a PhD you have to work independently. On the one hand, I totally agree. You have to work like a marketing person to position your research; like an engineer to implement your idea; like a sales representative to present your results; and like a manager to make a decision, and take responsibility if the idea does not succeed as expected.

On the other hand, I realize that I cannot make this thesis possible without support from the people around me. First, I am very grateful to my supervisor, Professor Dr. Uwe Schwiegelshohn, for giving me so much trust and assistance. Many thanks go further to all colleagues in my department, especially to Joachm Lepping for not only sharing the office but also giving useful comments and ideas. Also, Jörg Platte has always been kind to help me use our computer cluster.

In addition, I would like to acknowledge NRW Graduate School of Production Engineering and Logistics for the research grant; and its former co-ordinators: Dr. Meni Syrou and Gundula Pläp.

Moreover, I would like to thank the people at ILOG, especially Dr. Philippe Laborie and Michael Heusch for their valuable suggestions and guidance to solve my technical problems.

Also, I really appreciate Professor Dr. Chris Beck for his advice during the Doctoral Programme in CP 2007. Without his sample codes it would take me much more time if I had to start programming from scratch. Thanks to Dr. Jan-Hendrik Bartels for providing his random instances.

Many thanks to my sisters in Thailand, without whom I could not stay here for so long and really concentrate on my work. Finally, I would like to express my utmost gratitude to my parents for their love and dedication.

Abstract

In the automotive industry, a manufacturer must perform several hundreds of tests on prototypes of a vehicle before starting its mass production. These tests must be allocated to suitable prototypes and ordered to satisfy temporal constraints and various kinds of test dependencies. To reduce costs, the manufacturer is interested in using the minimum number of prototypes.

We apply Constraint Programming (CP) and a hybrid approach to solve the scheduling problem. Our CP method can achieve good feasible solutions even for our largest instances within a reasonable time. In comparison with existing methods, we can improve the solutions for most of our instances and reduce the average number of required prototypes. The hybrid approach uses mixed integer linear programming (MILP) to solve the planning part and CP to find the complete schedule. Although the hybrid approach is not as robust as CP with respect to data characteristics and additional constraints, it can complement CP in finding a better lower bound.

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

a_i	Construction time of prototype i
a'_i	Total construction time of prototype i
a_p	Constant period to construct a new prototype
b_v	Delivery delay of a component for prototype variant v
C_j	Completion time of test j
C_{max}	Makespan
$\bar{D}$	Set of due dates of all tests
$\bar{H}$	Total time horizon
I	Set of prototypes
J	Set of tests
J_{Last}	Set of crash tests
m	Number of prototypes
m_r	Number of required prototypes
$\bar{m}_r$	Lower bound of the number of required prototypes
M_j	Set of prototype variants that can perform test j
n	Number of tests
N_v	Set of tests which can be done by prototype variant v
p_j	Processing time of test j
r_j	Release date of test j
s_v	Set-up time of prototype variant v
$\bar{R}$	Set of release dates of all tests
V	Set of prototype variants
V'	Set of prototype variants including a dummy variant 0
$j \prec k$	Test j must be completed before test k is started
$j \sim k$	Tests j and k must be executed on the same prototype
$j \asymp k$	Tests j and k must be executed on different prototype
$j \simeq k$	Test j must precede test k or both tests must be executed on different prototypes

Chapter 1

Introduction

We start with the motivation of the test scheduling problem before describing the goals and the structure of the thesis.

1.1 Motivation

In the automotive industry, a manufacturer must typically carry out several hundred tests on vehicle prototypes before mass production of this vehicle can start. As a suitable production system has not yet existed, these prototypes are handmade and expensive (between 0.5 to 1.5 million euros each). Therefore, it is in the interest of the manufacturer to reuse the prototypes for several tests whenever possible. To this end, the tests must be arranged in an appropriate order. For instance, any crash test must obviously be the last test on this prototype. Ignoring cost differences between different variants of prototypes, the manufacturer wants to minimize the number of prototypes required for the testing process while not delaying the start of production.

Each prototype is a combination of various vehicle components, like the engine or the gear box. Usually, there are several types of most components. Due to various incompatibilities between different types of components, in practice, there are up to 600 possible variants with different costs and production times. We can only allocate a test to a prototype variant if this variant satisfies the component requirements of this test. For instance, while a prototype with a gasoline engine is not suited to execute a diesel engine test it does not matter which kind of engine is used to perform a brake system test. Therefore, it is necessary to build appropriate variants of prototypes such that the scheduler can assign all tests to their suitable prototypes.

Furthermore, we must consider various temporal restrictions that can be modeled as release and due dates. For instance, consider a driving test in wintry conditions. As the complete testing process typically takes more than one year, we can obtain the desired conditions for this test by selecting appropriate release and due dates. Also the manufacturer may use due dates to ensure that some critical tests are completed early enough to allow a repetition of the tests in case of failure.

Most companies have a special shop for producing prototypes. Due to the limited capacity of this shop, the prototypes are sequentially manufactured resulting in an availability time for each prototype. Following the manufacturing, a prototype requires an initial set-up process whose duration depends on the selected variant of the prototype and the planned test sequence on this prototype. For instance, a prototype for a corrosion test must be painted while this is not necessary for executing a crash test. Hence, the scheduler must ensure that tests can start only when the assigned prototype is constructed and sufficiently prepared for the test. Moreover, a valid schedule must obey additional constraints. For instance, a brake system test must be completed before starting a long-run test. Also, we cannot execute two long-run tests on the same prototype. Finally, a brake system test and a suspension system test must be executed on the same prototype in order to verify the co-ordination of both systems.

Nowadays computer simulation techniques have been applied to perform various kinds of tests like heat transfer simulation in a passenger compartment, calculation of flow patterns for wind-tunnel tests, or crash simulation. However, Kohlhoff [32] mentioned that the use of real prototypes is still indispensable for real verification because of the limitation in the accuracy of computation methods and models. Moreover, several case studies [6, 53, 40, 59] have been conducted with car manufacturers to develop a decision support tool for the test scheduling process, although their problem-specific requirements can be slightly different from our case, as discussed in Section 2.1. Therefore, it is important to develop an efficient scheduling approach to meet the complex requirements and reduce the number of required prototypes.

In practice, mostly heuristic algorithms are applied either in a stand-alone fashion or to generate a good initial solution for other complicated techniques [6, 53]. However, it takes a significant amount of time to develop good heuristics. Moreover, it is rather complicated to modify such an algorithm even if the problem is only slightly changed. Therefore, we are particularly interested in using a standard approach that has a comparable efficiency and robustness with respect to both the problem size and additional constraints. Thus, the approach will allow us to incorporate a later modification of the problem and or to improve the performance by fine tuning the algorithm.

In this thesis, we focus on the use of two new techniques: Constraint Programming (CP) and a hybrid approach based on Bender's Decomposition. CP has been successfully applied to solve various kinds of scheduling problems, as shown by Baptiste et al. [4]. CP has a flexible declarative programming language for a modeling part to generate a set of constraints to be satisfied; and a search part to describe how to search for solutions. Furthermore, CP applies a mechanism called constraint propagation to reduce the domain of variables. Both the reduction and the search parts must simultaneously interact to determine a solution.

In general, the hybrid approach is a combination of different classical methods like Mixed Integer Linear Programming (MILP), CP, or local search. It becomes a promising way to solve larger and more complicated problems. In many cases, the hybrid approach can achieve better solutions than pure algorithms based on only one technology, see Danna and Le Pape [15].

To apply the hybrid approach for our problem, we decompose our scheduling problem into two sub-problems. The solution to the planning or master problem is the minimum number of required prototypes and their corresponding variants such that the scheduling or slave problem can actually perform all tests. The master and slave problems are solved by MILP and CP, respectively, since MILP is a suitable tool for optimization, while CP has a capability to quickly find a feasible solution.

During the procedure, the MILP and CP models must complement each other. The optimal solution is achieved when CP manages to find a complete schedule corresponding to the solution of the planning problem solved by MILP. However, if CP fails to find any feasible solution, the planning problem must be reformulated such that the initial solutions to the previous planning problem are excluded. To remove those solutions, a Bender's cut constraint is generated and augmented to the master problem. The whole problem is proved to be infeasible if Bender's cuts eliminate every feasible solution in the MILP problem.

1.2 Goals of the Thesis

Our primary objective is to develop a solving approach which can schedule the given tests subject to all requirements using the minimum number of prototypes.

The approach has to determine several decisions: the number of required prototypes, the sequence of prototype variants, the allocation of tests to prototypes, and the sequence of tests on each prototype.

Moreover, the following aspects should be concerned:

- Scalability
  The approach should be applicable not only to a small size problem, but also to a practical size one of several hundreds of tests.
- Performance
  The approach should be able to find a good feasible solution within a reasonable time. An optimal solution should be found if the problem size is small enough.
- Robustness
  The approach should be robust to the changes of data characteristics and some problem-specific constraints.
- Agility
  The approach should be convenient for project implementation and later modification.

1.3 Structure of the Thesis

The thesis is organized as follows. First, Chapter 2 provides a general overview to some similar projects related to the test scheduling problem in the automotive industry including their solving techniques. As our problem can be classified as the resource investment problem, we also provide relevant literature reviews in this field of Operations Research.

Next, we formally describe our problem using the mathematical notations. In addition, we show the characteristic of the real-life instances obtained from a car manufacturer. We solve these data sets later by several approaches to carry out the preliminary study.

In general, there are many approaches that can be applied to solve a scheduling problem. Chapter 3 briefly provides a basic concept, advantages and disadvantages of several classical methods before describing the constraint-based scheduling and the hybrid approaches in detail.

Afterward, we formulate the MILP model to solve the whole problem in Chapter 4. Also, we suggest a simplified model to determine the lower bound of our large instances. The model considers two important principles: set covering to select prototypes with suitable components; and energetic reasoning to determine the number of prototypes required over the time intervals between the distinct values of the release and due dates.

In Chapter 5, our problem is formulated with the help of the CP approach. To enhance the performance of CP, we implement our own search strategy instead of using the available search strategies in our CP solver. Our strategy tries to find and allocate a critical test first, while using the minimum number of prototypes. Moreover, the flexibility of the CP approach allows us to further provide two trade-off analyses. Despite our original goal, we may allow to build more prototypes than necessary in order to reduce the total completion time of the testing process. On the contrary, when forcing to use less prototypes, we maximize the number of tests which can be executed on time.

Chapter 6 describes our hybrid approach. While the formulations for the master and slave problems are derived from our simplified MILP and CP models, we implement several kinds of Bender's constraint trying to avoid more invalid solutions in each iteration. Also, we suggest alternative formulations of the hybrid method resulting from letting CP to take more scheduling responsibility, while stronger Bender's cuts can be further incorporated.

As the number of our real-life instances is limited and not enough to verify the performance and robustness, we also apply our approaches to solve more random instances in Chapter 7. Bartels and Zimmermann [6] generated these instances to verify the performance of their heuristic approach in the case of another manufacturer. Although the new problem is slightly different from our case, we decided to use these instances in order to compare the results obtained by different methods and to study the robustness of our approaches to the changes in the characteristics of data and in the constraints of the problem.

Finally, Chapter 8 summarizes the thesis and outlines the potential directions for future research.

Chapter 2

Test Scheduling Problem

In this chapter, we discuss the test scheduling in detail. First, we provide a literature review of relevant problems in the automotive industry and in operations research. Also, the problem description is formally explained using the mathematical notations. Finally, we present the characteristics of the real-life instances which will be solved in the following chapters by various approaches.

2.1 Literature Review

We present the projects related to the test scheduling problem in several automobile manufacturers with their problem characteristics and the solving approaches. In fact, this test scheduling problem belongs to a class of the resource investment problem. A survey of relevant researches in the field of operations research is provided subsequently.

2.1.1 Review of the Test Scheduling Problems in the Automotive Industry

To solve our prototype problem, Scheffermann et al. [53] developed a heuristic approach based on specific problem knowledge and on some intuitive ideas. They improve the quality of their initial result by tuning some parameters with the help of statistical methods. This approach is used as a decision support tool to help the planning department in a real working environment. However, it turned out to be rather difficult to find an appropriate parameter set that leads to a good result.

Also, Karadgi [31] applied MILP to solve the simplified problem neglecting the component requirements and the variant selection process. He suggested two formulations to minimize the makespan; however, only small size instances of up to around 60 tests can be solved.

Later, Limtanyakul and Schwiegelshohn [38] applied CP to minimize the makespan of our problem while the number of prototypes is given as an input parameter. Although minimizing the makespan is not necessarily the primary objective, the manufacturer prefers to complete the testing process as soon as possible using the given number of prototypes. With this approach we can iteratively reduce the number of prototypes and repeat the procedure to find the minimum number of prototypes that still allows a valid schedule while observing the specified start of mass production. If the specified number of prototypes is large enough we can always find feasible or even optimal solutions. Also, we can prove that the problem is infeasible if the given number of prototypes is rather small. Therefore, we often encounter a large set of numbers of prototypes for which our CP approach can neither prove feasibility nor infeasibility of the problem within a reasonable time. We call this set *the problem gap*.

To complement the CP approach, Limtanyakul [36] suggested a simplified MILP model to find a better lower bound of the number of required prototypes. The results of this study confirm that CP has achieved good feasible solutions since only small gaps remain. Also, Limtanyakul and Schwiegelshohn [37] presented a preliminary idea of using a hybrid method where CP and MILP can work closely together in order to close this gap.

In addition to these previous studies, there are several projects related to vehicle tests of car manufacturers. They slightly differ in problem characteristics and are solved using various methods. However, they all have the same goal to reduce the cost of the testing process.

For the Ford Motor Company, Lockledge et al. [40] applied a multi-stage mathematical programming model to optimize the fleet of prototypes. In the first stage, they determine the number of required variants subject to component requirements of all tests. The second model determines the minimum number of cars of each variant such that all tests can be executed before their due dates. It is possible to apply MILP in this case because there are only few different values of due dates. Therefore, the problem reduces to an assignment problem of tests to one of these time slots. This assumption obviously simplifies the model and reduces the number of integer variables. Unfortunately, this method is not applicable to our case as the values of our due dates are widely spread over the whole time scale. Moreover, the complexity of our model quickly increases if we consider other temporal constraints like release dates and availability times.

Bartels and Zimmermann [6] dealt with a problem that is very similar to our problem. It also considers component requirements and temporal constraints, while some additional constraints are slightly different. Instead of tests being executed on the same or different prototypes, they consider partially ordered destructive tests. For instance, a driving test that can damage the chassis such that this prototype is no longer suitable to perform an acoustic test. Therefore, either this driving test is executed after the acoustic test or they are allocated on different prototypes. They presented an MILP formulation which can be applied to solve only small size problems. To handle larger instances, they propose a heuristic approach based on a priority-rule.

Furthermore, Zakarian [59] developed an analysis model and a decision support tool to evaluate the performance of product validation and test plans for General Motors Truck Groups. His work concentrated on stochastic scheduling. He modeled uncertainties associated with processing times of tests and product failures in order to determine the trade-off between the number of vehicles used in the validation plan and percentage of completed tests. First, he applied several heuristic rules to generate initial schedules based on his probability function. After that, he measured the performance of the obtained schedules with the help of a Markov model and simulations.

2.1.2 Review of the Resource Investment Problem in Operations Research

Notice that in standard scheduling problems [48], the number of machines or resources is typically known. However, the objective of our problem is to minimize the maximum demand of resource utilization. We must consider both resource planning and scheduling. Therefore, our problem is related to the resource investment problem (RIP) also known as the problem of scarce time. There jobs can simultaneously occupy several kinds of resources and must be finished within an overall project deadline. Assuming an unlimited capacity of resources, we want to achieve a non-delay schedule which minimizes the total resource cost. Möhring [43] and Demeulemeester [17] presented exact algorithms for solving small instances of the RIP. To deal with larger problems, Yamashita et al. [58] proposed a scatter search procedure with multi-start heuristics. Neumann and Zimmermann [45] suggested a heuristic algorithm for a resource leveling problem that includes RIP as a special case. Also, Caramia [12] suggested a local search algorithm to minimize the peak demand.

More generally, Nübel [46] studied the variant RIP/max by including the minimum and maximum time lags between the start time of activities. He applies a depth-first branch-and-bound algorithm to explore the search scheme generated from pseudo-semi active schedules. Hsu and Kim [23] suggested a priority-rule heuristic approach for the multi-mode resource investment problem (MMRIP) in which at least one of the activities can be undertaken in any of its several modes like having machines in parallel. Our test scheduling problem may also be classified as MMRIP/Max.

2.2 Formal Problem Description

After informally explaining the problem in the Section 1, we now introduce the mathematical notations used in the rest of this report. These notations are based on the book of Pinedo [48] whenever possible.

V, I, and J are the sets of prototype variants, prototypes, and tests, respectively, with $|V| = l$, $|I| = m$, and $|J| = n$. Additionally, we define $V' = V \cup \{0\}$ as the extended set of prototype variants including a dummy variant 0.

$M_j \subseteq V$ is the set of prototype variants that can perform test $j \in J$. $N_v \subseteq J$ represents a set of tests which can be executed by prototype variant v. Each prototype $i \in I$ corresponds to variant $v_i \in V'$. When $v_i = 0$, it means prototype i needs not be actually built since it cannot execute any test, that is, $0 \notin M_j$.

Each test $j \in J$ has a processing time p_j, a release date r_j, and a due date d_j. As all tests are executed without interruption, the completion time C_j of test j must obey $r_j + p_j \leq C_j \leq d_j$. The makespan is defined as $C_{max} = \max_{j \in J}\{C_j\}$.

Let define parameter $\bar{H}$ as a total time horizon of the whole schedule. It can be any large positive number which is expected earlier to be greater than the due dates of all tests. We specify $\bar{H} = \max_{j \in J}\{d_j\}$ for all computations in this thesis.

Also, we define set $\bar{R} = \{\bar{r}_1, ..., \bar{r}_{n_r}\}$ which contains the distinct elements in the array of all release dates $[r_1, ..., r_n]$. For instance, if our array of release dates is $[2, 0, 0, 7]$, we obtain $\bar{R} = \{0, 2, 7\}$. Similarly, set $\bar{D} = \{\bar{d}_1, ..., \bar{d}_{n_d}\}$ contains only the distinct elements in an array of all due dates $[d_1, ..., d_n]$.

Moreover, we consider the following relations between two different tests $j, k \in J$:

- $j \prec k$ iff test j must be completed before test k is started.
- $j \sim k$ iff tests j and k must be executed on the same prototype.
- $j \asymp k$ iff tests j and k must not be executed on the same prototype.

Finally, $J_{Last} \subseteq J$ is the set of tests that must not be followed by any other test on the same prototype.

Furthermore, we must formulate the availability time of a prototype. As already mentioned, we use a simple model for this purpose. This model considers the manufacturing time of a prototype, the set-up time of a prototype variant and a potential delivery delay of a component of this prototype.

We assume that the workshop requires a constant construction period a_p to build one new prototype. It means prototype $i \in I$ is not available before time $a_i = ia_p$. An additional time s_v is further required to set-up variant $v \in V$.

Occasionally, there may be a delay for the delivery of a component such that this delay cannot be compensated within prototype manufacturing. In this case, we assume that the delay time b_v of prototype variant v dominates the manufacturing of the prototype and includes the integration of this component into the already existing prototype. Therefore, a test cannot be executed on prototype i before the total prototype construction time $a'_i = \max\{a_i + s_{v_i}, b_{v_i}\}$.

As already mentioned, we assume that the costs of the various prototype variants are roughly similar. Therefore, our main objective is to minimize the number of required prototypes m_r, whose lower bound is represented by $\bar{m}_r$. Notice that $m_r \leq m$ as we need not use all given prototypes.

To match the common notation of scheduling problems, we consider prototypes and tests as machines and jobs, respectively. Using the standard scheduling framework, this problem is an extension of scheduling identical machines in parallel. We may use the notation $P_m|r_j, d_j, M_j, prec|m_r$ to represent the problem. However, note that this notation does not include all facets of the problem like the selection of a variant for each machine (prototype), as well as the application-specific conditions like performing two tests on the same prototype $(j \sim k)$.

Regarding the problem complexity, this optimization problem is equivalent to searching for the minimum number of machines which can lead to a valid feasible schedule that observes our due dates. If we omit temporal constraints, machine eligibility, and other additional requirements, our problem can be considered as a bin packing problem, that is, we try to find the minimum number of bins (or machines) to store items (or jobs) with various sizes (or processing times). Hence, our problem is NP-hard in the strong sense as it is a generalization of the bin packing problem, see Garey and Johnson [20].

Finally, as summarized in Figure 2.1, the scheduler has the following decision tasks:

1. Select the number and variant of prototypes from the given list

2. Allocate tests to each prototype

3. Order the sequence of tests on each prototype

2.3 Characteristics of the Real-Life Instances

From a real-life test scenario we obtain four data sets of different sizes from about 40 tests to almost 500 tests. Table 2.1 shows the characteristics of those data sets, like the number of variants and the average processing times of tests.

Note that the initial slack time of a test is defined as $d_j - r_j - p_j$, with the corresponding ratio $(d_j - r_j - p_j)/p_j$. However, r_j, d_j, and the slack time can be changed later due to the constraint propagation when CP starts solving the instance. We also define the variant ratio $VR = (\sum_{j \in J} |M_j|)/nl$.

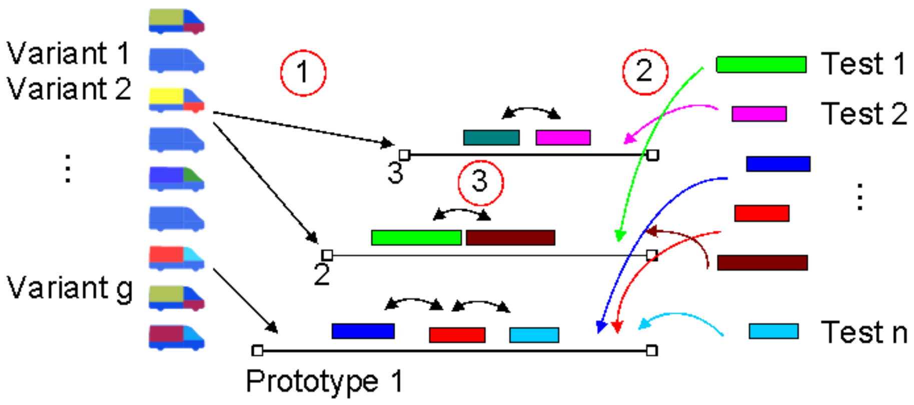

Figure 2.1: Tasks of the scheduler

These parameters can roughly measure the hardness to schedule the tests. We notice that the tests in the fourth instance have longer slack times compared to the other instances. Also, in the second instance, the variant ratio is almost one as most of the tests can be allocated to any prototype variant.

We further provide more details like the number of distinct values of release ($|\bar{R}|$) and due dates ($|\bar{D}|$), and the number of tests with precedence constraints. It can be seen that up to 10% of all tests may be subject to special constraints. In the large data set involving several hundred tests, the impact of those constraints may be considerable. Therefore, it is difficult to later verify and correct a schedule that has been obtained by neglecting these requirements.

Furthermore, we provide the total list of prototype components which must be considered in order to determine the machine eligibility set M_j.

- Engine type (Motortyp)
- Engine construction stage (Motorbaustufe)
- Gearbox type (Getriebetyp)
- Gearbox construction stage (Getriebebaustufe)
- Tonnage
- Wheelbase (Radstand)
- Specification (Ausführung)
- Height of storage area (Laderaumhöhe)
- Payload (Pritschenlänge)
- Steering (Lenker)

Finally, we determine the set-up time of prototype variants from the sum of the durations of these following steps.

Table 2.1: Detailed information of the real-life instances.

Instance	1	2	3	4
Number of tests	41	100	231	487
Number of variants	38	3	9	663
Average processing times (day)	11	10	10	45
Average release dates (day)	125	53	173	81
Average due date (day)	281	175	275	395
Average initial slack times (day)	145	113	92	274
Average initial slack ratio	20.32	29.08	21.52	41.78
Prototype construction period (day)	5	5	5	1
Average value of variant set-up times (day)	13	4	5	20
Average value of component delay times (day)	13	89	5	85
VR	0.36	0.98	0.31	0.02
$\lvert\bar{R}\rvert$	19	17	27	7
$\lvert\bar{D}\rvert$	25	18	41	74
$\lvert\{j \in J \mid j \prec k\}\rvert$	1	1	3	1
$\lvert\{j, k \in J \mid j \sim k\}\rvert$	2	2	6	68
$\lvert\{j, k \in J \mid j \asymp k\}\rvert$	2	6	6	2
$\lvert J_{Last}\rvert$	1	-	3	1

- Body work (Rohbau)
- Painting (Lackierung)
- Installation (Montage)
- Interval (Abstand)
- Commissioning (Inbetriebnahme)

Chapter 3

Methodology

The aim of this section is to provide a general overview of approaches currently applied to solve a scheduling problem. First, we describe four classical approaches: problem-specific operations research algorithms, Constraint Programming, Mixed Integer Linear Programming, and local search.

To apply CP for solving scheduling problems, we also provide the common notations and the concepts of standard propagation rules. Finally, we generally introduce the hybrid approach which is the combination of different classical methods.

3.1 General Overview of Classical Solving Approaches

These classical approaches emerged from different areas: operations research, mathematics, and computer science. Originally, each method is applied individually to solve a specific problem. It is found out later that in many situations the same scheduling problem can be represented and solved by another kind of technology. In the following, we briefly summarize basic concepts, advantages, and disadvantages of each method.

- Problem-Specific Operations Research Algorithms
  This area of study has been actively developed for several decades from industrial engineering or management sciences. It defines a scheduling problem as the allocation of tasks (or jobs) to scarce resources (or machines) over time. A scheduling problem is represented by three elements to describe the machine environment, constraint, and objective [48]. Jobs are scheduled such that the objective is optimized subject to some given constraints. Many kinds of problems are studied and analyzed thoroughly. Researchers have been successfully developing algorithms to solve them optimally or provide near-optimal solutions within a time bounded by a polynomial function of the problem size. However, each algorithm is designed just for solving a specific problem. When the problem is slightly changed or includes some additional constraints, the algorithm often leads to a poor solution.

- Mixed Integer Linear Programming (MILP)
  MILP is a well known method to solve many combinatorial optimization problems [44, 57, 56]. Recently, many developments in theory have been integrated into MILP software tools, for example cutting plane methods, branch strategy, and indicator constraints [24]. The progress of applying MILP was reported by Bixby et al. [9]. However, this method

requires a significant amount of experience to formulate a mathematical model for each problem. Moreover, Linderoth and Ralphs [39] mentioned that the solving performance strongly depends on the used solvers.

In general, the branch-and-bound mechanism is applied to find a solution by relaxing some constraints, for instance, integrality constraints. It becomes easier to optimally solve the resulting problem. The solution is a lower bound which can be used to prune a domain of the search tree. The lower bound can further be improved by branching, that is, splitting the feasible region into smaller parts. Alternatively, we can generate cutting planes, which are linear inequalities added to the relaxation problem in order to remove the optimal LP solution, while at least one optimal integer solution is kept.

MILP is particularly efficient if the continuous relaxation can be iteratively tightened by adding cutting planes to improve the approximation. The approach is not robust to variations in the problem size as the size of an MILP formulation typically grows much faster than the actual size of the problem. Furthermore, many researchers report that MILP alone is not an effective tool to solve large-scale scheduling problems for parallel machines with release and due dates. For instance, Hooker [21] and Sadykov and Wolsey [52] compared the performance of several methods including the MILP approach when minimizing makespan and allocation cost. Their results showed that MILP can handle small problems well but that it is not appropriate for cases dealing with many jobs like our problem.

- Constraint Programming (CP)
  CP is another deterministic approach for formulating and solving an optimization problem. It was originally developed for computer science applications. After a few decades, CP has widely appeared in different research areas, for instance, planning and scheduling, bioinformatics, vehicle routing, and configuration [51].

  To apply CP, we first describe a problem formulation as a set of decision variables and constraints. Each variable has its own finite set of possible values (domain), while constraints restrict the value that variables can simultaneously take. A solution to CP is an assignment of one value to every variable such that all constraints are satisfied. Next, we can specify a search scheme to describe how the solver enumerates through the possible assignments in order to achieve a solution. When a contradiction occurs during the search, CP has a functionality called backtracking to keep trying other possible decisions.

  Since the search space is normally so large, CP also uses the constraints to remove inconsistent values from the domain of variables. Each constraint has corresponding filtering algorithms. The filtering algorithms can be applied initially before starting the search and also simultaneously. Moreover, every time when the domain of a variable is changed, it will trigger the filtering algorithms of constraints involving that variable. This can lead further to the change in the domains of other variables and again trigger their filtering algorithms. Thus, the process is known as constraint propagation.

  Until now, CP is applied only to find a feasible solution to the standard Constraint Satisfaction Problem. For solving an optimization problem, CP has a procedure to add a new constraint such that the new objective is strictly better than the current value. As a result, the solution will be improved until it reaches the optimal value.

CP is superior to MILP in expressing constraints that are not limited to linear inequalities. For instance, it supports logical expressions, like *if...else*, instead of using many complex linear inequalities to represent the machine eligibility constraints. This leads to a natural formulation of our complex scheduling problem.

In addition, we can enhance the performance of the CP solver by providing a problem specific search strategy. The experience from using heuristics can be included into the CP framework. Puget and Lustig [50] as well as Milano and Trick [42] provided more reviews about the comparison between both MILP and CP in view of modeling and solving techniques.

CP can solve a problem efficiently if constraints propagate well and tighten the objective value. Therefore, minimization of the makespan is well suited as the last completion time of all jobs directly leads to a lower bound of the objective value. For minimizing a sum of set-up times or the allocation cost, any bounds of the cost function do not indicate directly which term of the sum should be reduced, see Danna and Pape [15].

- Local Search
  In many cases, it is not practical to apply a deterministic approach, especially when the size of a problem becomes larger, as most of the scheduling problems are NP-hard, see Bagchi [3]. The local search method can be a good alternative in case that good feasible solutions are acceptable and should be found within a short period.

  Although in many cases local search algorithms can converge quickly and find a good solution, the trivial bounds on time complexity cannot be derived mathematically. Another major disadvantage is that it cannot guarantee to provide an optimal solution or prove infeasibility.

  This method has been used for several decades in combinatorial optimization. The basic idea of local search is to find improvements to the current solution by making small changes. At the beginning, an initial solution has to be given. After that, the algorithm tries to optimize the solution by moving around the space of candidate solutions (the search space) until a solution seems to close an optimal value or a time bound is finished. There are various kinds of local search, for instance, simulated annealing or tabu search.

  Apart from the basic local search, Genetic Algorithms are developed in order to explore the neighborhood more globally, see Aarts et al. [1]. The basic concept of the approach involves a population of solutions, processes of natural selection, mutation and recombination to produce better solutions.

3.2 Constraint-Based Scheduling

CP has been further applied to specifically handle a scheduling problem. We can observe this method for many real-life applications, like production scheduling in steel manufacturing [16], job-shop scheduling [47], and railway scheduling for cane transportation [41]. The success is due to the development of efficient constraint propagation rules for scheduling problems in several scenarios like disjunctive (or cumulative) resources with (or without) preemption, see Baptiste et al. [4]. Moreover, the functionality to model and solve scheduling problems has been included in several famous CP solvers like ILOG Scheduler [26], ECLipSe [14], and CHIP V5 [7].

Although our scheduling problem is just a specific case, we introduce the common notations used to represent scheduling problems in general. Thus, we can later have a better understanding of the mechanism of constraint propagation.

3.2.1 Representation of Scheduling Problems in CP Framework

We can consider a scheduling problem as a constraint satisfaction problem or as a constraint optimization problem. First, we define a set of activities which are tasks or work to be completed. All activities must then be allocated over time to a set of resources, for instance, workers, machines, or raw materials.

Note that the activities and resources are similar to jobs and machines used normally in the standard scheduling problem. However in CP, these notations have more general definitions in order to ease the formulation of the complex problem.

Also, we introduce two kinds of constraints: resource constraints due to the limited capacity of the resources; and the temporal constraints to define possible relationships between the start and end times of the activities.

3.2.1.1 Resources

There are two types of resources: a disjunctive resource which can execute at most one activity at a time; and a cumulative resource which can perform several tasks in parallel if the resource capacity is not exceeded. Although the disjunctive resource can be represented by the cumulative resource with a unit capacity, CP can apply more restricted and efficient propagation rules for the disjunctive resource.

Moreover, we can construct more complex scheduling problems like parallel machines from a set of disjunctive resources. It is also possible to use one cumulative resource with capacity equal to the number of machines. However, we may get the solution in which jobs can be transfered to different machines during their execution. Nevertheless, the cumulative representation is preferred in appropriate applications since it can avoid the exploration of symmetrical configurations and reduce the search space [25].

3.2.1.2 Activities

Each activity has a duration to be processed on the resource. A non-breakable activity is an activity which executes without preemption from its start time to its end time, while a breakable activity can be interrupted during the resource breaks, like weekend or lunch, but not for processing other activities.

Also, in the case of elastic scheduling, the resource consumption of the activity can be varied over time between 0 to the resource capacity. The sum of the resource consumption over time must be equal to the given energy. For instance, an activity which requires the energy of 8 units can be executed with a duration of 4 and a resource capacity of 2, or a duration of 2 and a resource capacity of 4.

In non-preemptive scheduling, each activity A_j has three variables: $start(A_j)$, $end(A_j)$, and $proc(A_j)$ to represent the start time, end time, and processing time. In fact, if the processing time is constant, it is sufficient to have one variable for each activity. Despite the redundancy, three variables are still preferred in order to simplify the description of the constraints, see Barták [5].

The time horizon must be discretized into identical slots. With the release date r_j and due date d_j, the initial domains of $start(A_j)$ and $end(A_j)$ are $[r_j, lst_j]$ and $[eet_j, d_j]$ where lst_j and eet_j stand for the latest start time and earliest end time of activity A_j. Figure 3.1 represents the domains of an activity with $p_j = 4, r_j = 0$, and $d_j = 10$.

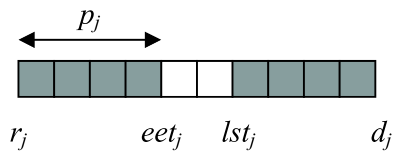

Figure 3.1: Activity and its time domains.

The representation of preemptive scheduling is more complicated as two more variables must be defined. $X(A_j, t)$ is a binary variable, which takes the value one iff A_j executes at time t. $set(A_j)$ is a set variable representing the set of times at which A_j executes. As a result, we can derive the following relations between these variables

$$\begin{aligned} X(A_j, t) = 1 &\Leftrightarrow t \in set(A_j) \\ start(A_j) &= \min_{t \in set(A_j)} t \\ end(A_j) &= \max_{t \in set(A_j)} (t + 1) \\ proc(A_j) &= |set(A_j)| \end{aligned}$$

In non-preemptive scheduling, we further have $set(A_j) = [start(A_j), end(A_j))$, while $|set(A_j)| = proc(A_j) = end(A_j) - start(A_j)$.

Furthermore, in some scheduling problems like parallel machines, the activities can be allocated to one resource from a set of compatible resources. We have to introduce a variable x_j representing the allocation of activity j to one of the machines in set I. Next, we consider each activity as being split into m fictive activities for all machines. After that, the constraint propagation process will deduce new time bounds for all fictive activities possible to utilize resource i. Whenever the time bounds of an alternative activity A_j^i are found inappropriate, the resource i cannot actually execute activity j. Thus, we can remove i from the domain of x_j.

3.2.1.3 Resource Constraints

As the activities compete for a resource over the time horizon, we need the resource constraints to ensure that the total consumption is not higher than the resource capacity. It is simple for the disjunctive resource as we can pose on a given machine either $end(A_k) \leq start(A_j)$ or $end(A_j) \leq start(A_k)$, $\forall j, k \in J, j \neq k$.

For a cumulative resource with elastic activities, we assume that an activity can take the resource amount $E(A_j, t)$ at any time t on the resource with the capacity C. Also, the activity has the energy requirement of $E(A_j)$ to be fulfilled. The resource constraints are given as follows:

$$\begin{aligned} \sum_t E(A_j,t) = 0 \quad &\Leftrightarrow \quad X(A_j,t) = 0 \\ \sum_t E(A_j,t) \quad &= \quad E(A_j) \\ \sum_{j=1}^{n} E(A_j,t) \quad &\leq \quad C \ . \end{aligned}$$

In non-elastic scheduling, we can include an additional constraint $E(A_j,t) = c_j X(A_j,t)$, where c_j represents the amount of resource required by activity A_j. This leads further to the subsequent conditions:

$$\sum_{j|start(A_j) \leq t < end(A_j)}^{n} c_j X(A_j,t) \leq C$$

for the preemptive scheduling and

$$\sum_{j|start(A_j) \leq t < end(A_j)}^{n} c_j \leq C$$

for the non-preemptive scheduling. Note also that the disjunctive resource is a special case of cumulative resource with $C = 1$.

The constraint propagation techniques for the fully elastic case can also be applied to the partially elastic and non-elastic problem. Similarly, we can apply the propagation rules from the preemptive to the non-preemptive scheduling and from the cumulative to disjunctive resource, see Le Pape [34]. All the relationships can be summarized in Figure 3.2.

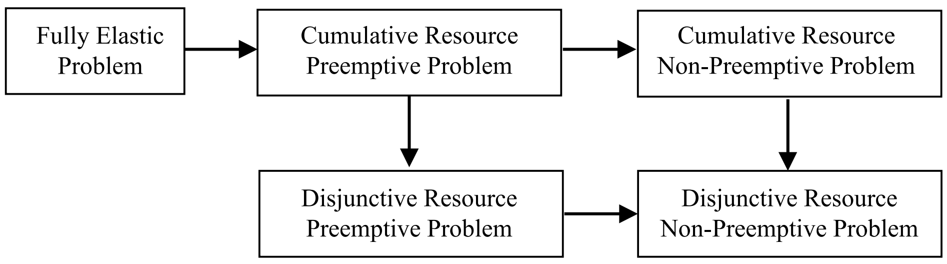
Fully Elastic Problem
Cumulative Resource Preemptive Problem
Cumulative Resource Non-Preemptive Problem
Disjunctive Resource Preemptive Problem
Disjunctive Resource Non-Preemptive Problem

Figure 3.2: Resource constraint propagation.

3.2.1.4 Temporal Constraints

There are two types of temporal constraints. The precedence constraint is used for defining a relation of start (or end) time between two activities. Time-bounded constraints specify that the start (or end time) of one activity must be after (or before) a given time.

3.2.2 Constraint Propagation

The constraint propagation process helps us not to waste time exploring the inconsistent nodes found by the filtering algorithms. However, if we spend too much effort on the propagation mechanism, it can take a large amount of computational time and memory. Thus, it is important to apply only the necessary rules. Here we provide the concept of the standard rules which are applied later in the thesis.

According to the characteristic of our problem, we introduce the principles of these algorithms, that is, a timetable constraint, a disjunctive constraint, and an edge-finding technique only in the case of the disjunctive resource and without preemption. Many variants of these rules for the cumulative resource with or without preemption are provided in Baptiste et al. [4].

Furthermore, the energetic reasoning is discussed. Although it is designed for the cumulative resource, the concept is relevant to our lower bound MILP formulation in Section 4.2.2.

All constraint propagation rules mentioned before are based on the time windows of activities. However, Laborie [33] suggested that these rules are quite inefficient especially when the time windows are very loose as they propagate nothing. We present also his new concepts based on the relative positions, that is, an energy precedence graph and a balance constraint.

3.2.2.1 Timetable Constraint

A data structure called Time-Table is basically applied to maintain information about resource utilization and resource availability over time. The constraints can be propagated in two directions. We can use information in the Time-Table of resources to change the time bounds (earliest start time and latest end times) of activities. Also, the Time-Tables must be updated when the time bounds of activities are changed due to any kind of constraint propagation.

Remember that we have the disjunctive resource constraint $\sum_{j=1}^{n} X(A_j,t) \leq 1$ and the non-preemption requirement $X(A_j,t) = 1$ iff $\mathit{start}(A_j) \leq t < \mathit{end}(A_j)$. The time bounds of activities can be changed by the following conditions:

$$\begin{aligned}
[X(A_j,t)=0] \wedge [t < \mathit{eet}_j] \quad &\Rightarrow \quad [\mathit{start}(A_j) > t] \\
[X(A_j,t)=0] \wedge [\mathit{lst}_j \leq t] \quad &\Rightarrow \quad [\mathit{end}(A_j) \leq t]
\end{aligned}$$

We can realize that $X(A_j,t)=0$ if at time t the resource is definitely occupied by other activities. As the preemption is not allowed, the activity cannot be executed and finish before its original earliest end time. Thus, the activity can start only after time t. Similarly, the activity must finish earlier if the resource is not available between its original latest start time and the end time. Also, when the time domains of activities are changed, we can update the information back to the time-table using

$$\begin{aligned}
\mathit{start}(A_j) \quad &\geq \quad \min\{t|\mathit{ub}(X(A_j,t))=1\} \\
\mathit{end}(A_j) \quad &\leq \quad \max\{t|\mathit{ub}(X(A_j,t))=1\}+1
\end{aligned}$$

where ub denotes an upper bound value of the variable. When $\mathit{ub}(X(A_j,t))=1$, it is still possible to execute activity j at time t.

3.2.2.2 Disjunctive Constraint

The disjunctive constraint ensures that two activities requiring the same machine cannot overlap in time. Either activity A_j precedes activity A_k or vice versa.

$$\mathit{end}(A_j) \leq \mathit{start}(A_k) \vee \mathit{end}(A_k) \leq \mathit{start}(A_j) \tag{3.1}$$

When there are n activities, the algorithm needs quadratic computation in order to maintain totally $n(n-1)/2$ disjunctive constraints. Also, whenever the algorithm detects that the earliest start time of activity j exceeds the latest start time of activity k, it can deduce that activity k precedes activity j. Thus, the time bounds of both activities can be tightened by a new temporal constraint $\mathit{end}(A_k) \leq \mathit{start}(A_j)$. A contradiction is detected if neither of the two activities can precede each other.

3.2.2.3 Edge-finding

While the disjunctive constraint analyzes the precedence relation between pairs of activities, the edge-finding technique consists of orderings of activities requiring the same resource. At each node in a search tree, let Ω be a set of the selected activities. The bounding technique tries to prove further that some activities must, can, or cannot execute first (or last) in Ω. This result can lead to new possible orderings and new time bounds.

Let $\tilde{r}_\Omega = \min_{k\in\Omega} \mathit{start}(A_k)$ be the smallest start time of the activities in Ω. Also, $\tilde{d}_\Omega = \max_{k\in\Omega} \mathit{end}(A_k)$ represents the largest end time of the activities in Ω. Finally, we define $\tilde{p}_\Omega = \sum_{k\in\Omega} p_j$ as the sum of the processing times of all activities in Ω.

Note that activity A_j cannot execute after any activity in Ω if the duration between the smallest start time of activities in Ω and the largest end time of activities in Ω including A_j is not enough to execute all activities in $\Omega \cup \{A_j\}$. The opposite case is when A_j cannot start before any activity in Ω. We represent both conditions as follows

$$\begin{aligned}
\tilde{d}_{\Omega\cup\{A_j\}} - \tilde{r}_\Omega < \tilde{p}_\Omega + p_j \quad &\Rightarrow \quad [A_j \ll \Omega],\ \forall\Omega, \forall A_j \notin \Omega \\
\tilde{d}_\Omega - \tilde{r}_{\Omega\cup\{A_j\}} < \tilde{p}_\Omega + p_j \quad &\Rightarrow \quad [A_j \gg \Omega],\ \forall\Omega, \forall A_j \notin \Omega
\end{aligned}$$

where $A_j \ll \Omega$ ($A_j \gg \Omega$) means A_j executes before (after) all activities in Ω, while $\tilde{d}_{\Omega\cup\{A_j\}} = \max_{k\in\Omega\cup\{A_j\}} \mathit{end}(A_k)$ and $\tilde{r}_{\Omega\cup\{A_j\}} = \min_{k\in\Omega\cup\{A_j\}} \mathit{start}(A_k)$. Moreover, we can compute the new time bounds

$$\begin{aligned}
[A_j \ll \Omega] \quad &\Rightarrow \quad \mathit{end}(A_j) \leq \min_{\emptyset\neq\Omega'\subseteq\Omega} (\tilde{d}_{\Omega'} - \tilde{p}_{\Omega'}),\ \forall\Omega, \forall A_j \notin \Omega \\
[A_j \gg \Omega] \quad &\Rightarrow \quad \mathit{start}(A_j) \geq \max_{\emptyset\neq\Omega'\subseteq\Omega} (\tilde{r}_{\Omega'} + \tilde{p}_{\Omega'}),\ \forall\Omega, \forall A_j \notin \Omega
\end{aligned}$$

All above constraints are used to determine whether activity A_j must execute first (or last) in a set of all activities. There are still other similar rules to detect whether activity A_j can or cannot execute first (or last) in a set of all activities, as described in [4].

To adjust the time bounds of n activities, an algorithm with the complexity of $O(n^2)$ was firstly implemented by Carlier and Pinson [13]. Later, they developed an $O(n \log n)$ algorithm; however, a more complex data structure is required [4].

3.2.2.4 Energetic Reasoning

The energetic reasoning is based on the comparison between the amount of energy which a resource can provide over a time interval $[t_1, t_2)$ and the amount of energy definitely required to process the activities. Here we present only the condition for a cumulative resource and without preemption.

We define $W_{Sh}(A_j, t_1, t_2)$ as the amount of energy definitely required by an activity A_j during a time interval $[t_1, t_2)$, and

- $p_j^+(t_1) = \max\{0, p_j - \max\{0, t_1 - r_j\}\}$ is the number of processing time units required by A_j if it is left-shifted, that is, scheduled as soon as possible;
- $p_j^-(t_2) = \max\{0, p_j - \max\{0, d_j - t_2\}\}$ is the number of processing time units required by A_j if it is right-shifted, that is, scheduled as late as possible.

As limited by the duration of time interval, A_j definitely consumes the resource with $W_{Sh}(A_j, t_1, t_2) = c_j(\min\{(t_2 - t_1), p_j^+(t_1), p_j^-(t_2)\})$. We can compute the overall energy consumption of all activities by $W_{Sh}(t_1, t_2) = \sum_{j \in J} W_{Sh}(A_j, t_1, t_2)$. As the resource can provide at most $C(t_2 - t_1)$, the schedule is still valid as long as $C(t_2 - t_1) - W_{Sh}(t_1, t_2) \geq 0$ for all t_1 and t_2 with $t_2 \geq t_1$.

Moreover, we can use the values of W_{Sh} to adjust the time bounds of activities. Since A_j needs the amount of energy at $c_j p_j^+(t_1)$ in order to be scheduled as early as possible, we can realize that it is not possible to finish A_j before t_2 if there is a time interval $[t_1, t_2)$ where

$$W_{Sh}(t_1, t_2) - W_{Sh}(A_j, t_1, t_2) + c_j p_j^+(t_1) > C(t_2 - t_1) \ .$$

Also, the new lower bound of the end time of A_j can be computed as

$$t_2 + \frac{1}{c_j}(W_{Sh}(t_1, t_2) - W_{Sh}(A_j, t_1, t_2) + c_j p_j^+(t_1) - C(t_2 - t_1)) \ .$$

Similarly, A_j is not allowed to start execution before t_1 if we find

$$W_{Sh}(t_1, t_2) - W_{Sh}(A_j, t_1, t_2) + c_j \min((t_2 - t_1), p_j^+(t_1)) > C(t_2 - t_1)$$

and the lower bound of the start time of A_j becomes

$$t_2 - \frac{1}{c_j}(C(t_2 - t_1) - W_{Sh}(t_1, t_2) + W_{Sh}(A_j, t_1, t_2)) \ .$$

The $O(n^3)$ algorithm to compute the time adjustment of n activities is shown in [4].

3.2.2.5 Energy Precedence Graph

The energy precedence graph is defined for a cumulative resource using a directed graph. Each node represents a resource event due to the change of the resource availability, for instance, the start of an activity. An edge is a precedence relation between a pair of two events. The precedence relations of events can be derived from several relations like initially-specified temporal constraints, search decisions, or the orderings discovered by other propagation algorithms. During computation, new events and new precedence relations can be incrementally inserted.

Based on the energy requirement, the algorithm maintains arc-consistency and deduces the time bound of these events.

Here, we demonstrate the idea using a simple scenario, while the formal representation can be found in Laborie [33]. Let x be a resource event and Ω be a subset of events which happen and finish before x. As the resource with capacity C must provide enough energy to execute all resource constraints in Ω, we can deduce a time bound of x by

$$start(x) \geq \min_{y \in \Omega}(start(y)) + \frac{\sum_{y \in \Omega} c_y p_y}{C(R)}$$

where c_y and p_y correspond to the consumption rate and the duration of event y.

The propagation of the energy precedence graph can be performed within the complexity of $O(n(\bar{p} + \log n))$, where n is the number of events and $\bar{p}$ is the maximum number of predecessors of a given event in the graph.

3.2.2.6 Balance Constraint

The balance constraint is another filtering algorithm based on relative positions. It maintains the minimum and maximum capacity requirements by determining the levels of resource consumption both before and after the start and end time of activities in the precedence graph. Although the balance constraint is generally defined for a discrete reservoir in which activities may produce or consume the resource capacity, it can also be applied to a discrete resource by adding a production event at the time origin of the schedule.

Laborie [33] provided the propagation rules which in fact are more beneficial for the problem with the discrete reservoir since the production events allow us to tighten the domain of time variables and determine new precedence relations between these time points.

In the case of the discrete resource, the balance constraints can be used only to detect a contradiction. Given an event x and the set of events $B(x)$ which necessarily start before x, a contradiction is found whenever the upper bound of resource level before x, $L^{<}_{\max}(x) = C - \sum_{y \in B(x)} c_y$, becomes less than zero.

The balance constraint should be used only for small or medium-size problems because it requires considerably high computation cost, that is, $O(n^3)$ for the worst case [26].

Laborie [33] mentioned that the energy precedence graph is based on global energy considerations like the edge-finding or the energetic reasoning algorithms, while the balance constraint is more similar to the timetable approach since we are interested in the level of resource demand. Moreover, all propagation rules are complementary since no technique can dominate the others.

3.3 The Hybrid Approach

Until now there has not been a single approach that can efficiently solve all kinds of combinatorial optimization problems. Recently, many researchers have tried to combine two different approaches together. Many cases show that hybrid approaches can achieve better results than algorithms based on only one technology.

To apply the hybrid approach, there are many possibilities that depend on the choice of two classical methods. Overviews of some important kinds of hybrid approaches are given by Danna et al. [15] and Aron et al. [2]

There are two types of cooperative schemes. In the decomposition scheme, one technique is applied to solve a sub-problem. Information received from solving this part is used by another technique to solve the overall problem. The multiple search scheme uses two or more optimization methods solving the full problems in turns or in parallel and exchanging information to improve the subsequent search.

However, it is still very difficult and challenging to develop the hybrid strategy for practical applications. First, we need to formulate several models instead of a single one. It requires much more time and effort during the modeling phase. Moreover, there are very few designers who know how to efficiently apply all these four classical techniques. They have to select methods that seem to be most suitable and then design a good coordination scheme. Otherwise, a considerable amount of time will be lost for interaction and communication between different schemes rather than for problem solving.

The combination between MILP and CP is one of the famous strategies in the area of hybrid optimization. There are many ways to combine MILP and CP. The simplest idea is developed, for instance, in a warehouse location problem [25]. MILP is applied to calculate the lower bound at each node of the search tree, while CP provides flexibility to express a compact model and a search procedure. As the model is mixed with the CP formulation, linear relaxation is applied only to all linear constraints.

Moreover, Thorsteinsson [54] proposed the branch-and-check framework, which integrates Bender's decomposition, branch-and-bound, and CP. The branch-and-price method was suggested by Easton et al. [19]. CP is applied to solve a pricing sub-problem and generate a column augmented to a master MILP problem. Apart from that, the branch-and-cut algorithm for MILP can be modified by using CP to detect infeasibility and generate cuts, as introduced by Bockmayr and Pisaruk [10].

In this thesis, we apply the MILP-CP hybrid approach based on Bender's decomposition. It was originally developed by Jain and Grossmann [29] to minimize the allocation cost. Later Hooker [22] used it to minimize the number of late jobs and total tardiness. Benini et al. [8] reported its application to minimize the communication cost in a Multi-Processor System-on-Chip.

Chapter 4

MILP Approach

In this section, we apply the MILP approach to solve our test scheduling problem. First, we formulate the complete MILP model to minimize the number of required prototypes with respect to all constraints. The model is based on sequencing variables and disjunctive constraints.

Also, we suggest a simplified MILP model to determine the lower bound of our large instances. To formulate the lower bound model, we apply the concept of set covering to consider the component requirements. Additionally, the impact of the temporal constraints can be partially considered after we further include the energetic reasoning in the model.

We finally provide the computational results of solving our real-life instances using the complete and simplified MILP models.

4.1 Complete Formulation

There are many possibilities to formulate an MILP model for solving an optimization problem. The modeling style has an impact on the problem size, that is, the number of variables and constraints. Also, it is more effective for the branch-and-bound mechanism if the LP relaxation model can refer to a good lower bound.

Bartels and Zimmermann [6] suggested an MILP formulation for their test scheduling problem. The model uses time-indexed variables. The time horizon is discretized into slots of identical size. Each slot requires n variables to represent which job occupies a machine at the moment. Thus, the size of the MILP model not only depends on the number of tests but also on the length of time horizon of the solving scheduling problem. That means the problem complexity can be significantly influenced by the characteristics of the input data.

Here, we develop our MILP model to describe our complete requirements and solve our real-life instances. Moreover, we rather use sequencing variables which represent the order of any two jobs since the size of the model can be limited to a function of the number of tests and prototype variants.

Assuming that all prototypes are identical, Karadgi [31] provided two MILP formulations based on sequencing variables. The first model uses w_{ijk} variables to represent both the allocation of test j to prototype i and its sequence with test k, while the second model uses x_{ij} and y_{jk} variables to separate the test allocation from the sequencing decision.

Also, the definition of the sequence of tests in both models is slightly different. The first model considers the sequence of tests j and k only when test j *immediately* precedes test

k. In the second model, the sequence is defined as when test j is scheduled *anytime* before test k. For instance, we consecutively execute test j, k, and l on machine i. That corresponds to the results $w_{ijk} = w_{ikl} = 1$ and $w_{ijl} = 0$ in the first model, while $y_{jk} = y_{kl} = y_{jl} = 1$ in the second model since test j just precedes test l, but not immediately.

Karadgi [31] mentioned that the second model requires about twice as many constraints as the first model. However, a number of decision variables in the first model tends to explode more abruptly when the problem slightly increases. The computational results also shows that the second model can solve a larger instance with around 60 tests.

Regarding the problem size, we formulate our problem with complete requirements using x_{ij} and y_{jk} variables like the second formulation. However, we need to introduce another binary decision variable z_{vi} to determine the relation to the variant of prototype i. In the following, we formally provide the definition of all variables:

- Test-to-prototype allocation variable

$$x_{i,j} = \begin{cases} 1 & \text{if test } j \text{ is allocated to prototype } i \\ 0 & \text{otherwise} \end{cases}$$

- Test sequencing decision variable

$$q_{j,k} = \begin{cases} 1 & \text{if test } j \text{ is scheduled \textit{anytime} before test } k \\ 0 & \text{otherwise} \end{cases}$$

  It should be clear that we can interpret the sequence of tests from the $q_{j,k}$ variable only when both tests j and k are executed on the same machine, that is, $x_{i,j} = x_{i,k} = 1$.

- Prototype variant decision variable

$$z_{v,i} = \begin{cases} 1 & \text{if prototype } i \text{ belongs to variant } v \\ 0 & \text{otherwise} \end{cases}$$

  Note that we may not require all prototypes of I. Thus, the ones not needed may remain unassigned, $\sum_{v \in V} z_{v,i} = 0$.

- Completion time variable

$$C_j \in \mathbf{N} = \{0, 1, \dots\} \quad \text{represents the completion time of test } j$$

Next, we can formulate our MILP model using the objective function (4.1) to minimize the number of required prototypes.

$$\text{Minimize} \quad \sum_{v \in V} \sum_{i \in I} z_{v,i} \tag{4.1}$$

For the allocation of tests and the selection of prototype variants, we use the following constraints:

$$\sum_{v \in V} z_{v,i} \leq 1 \qquad \forall i \in I \tag{4.2}$$

$$\sum_{i \in I} x_{i,j} = 1 \qquad \forall j \in J \tag{4.3}$$

$$\sum_{v \in V} z_{v,i} \geq x_{i,j} \qquad \forall i \in I, \forall j \in J \tag{4.4}$$

$$z_{v,i} + x_{i,j} \leq 1 \qquad \forall i \in I, \forall j \in J, \forall v \in V' \cap M_j. \tag{4.5}$$

Constraint (4.2) enforces that prototype i is assigned to at most one prototype variant. When prototype i is not required to execute any test, it will not be assigned to any variant. That means $\sum_{v \in V} z_{v,i}$ will be forced to remain zero due to our objective function (4.1).

Test j must be allocated to exactly one prototype by Constraint (4.3). Constraint (4.4) states that only a prototype whose variant has already been selected can execute a test. Due to the component requirement, Constraint (4.5) restricts test j from executing on prototype i which does not have suitable components.

After that, we define the disjunctive constraints:

$$q_{j,k} + q_{k,j} = 1 \qquad \forall j,k \in J; j \neq k \tag{4.6}$$

$$C_j + p_k - \bar{H}(3 - q_{j,k} - x_{i,j} - x_{i,k}) \leq C_k \qquad \forall i \in I; j,k \in J; j \neq k \tag{4.7}$$

$$C_k + p_j - \bar{H}(2 + q_{j,k} - x_{i,j} - x_{i,k}) \leq C_j \qquad \forall i \in I; j,k \in J; j \neq k \tag{4.8}$$

$$q_{j,j} = 0 \qquad \forall j \in J. \tag{4.9}$$

Constraints (4.6) ensures that either test j precedes test k or vice versa. Remember that such a relation can be interpreted only when both tests are allocated to the same prototype. For tests using the same prototype, we have to ensure that they occupy the resource at different times. When $j \prec k$, constraint set (4.7) is valid and constraint set (4.8) is redundant. Similarly, when $k \prec j$, constraint set (4.8) is valid and constraint set (4.7) is redundant. Furthermore, Constraint (4.9) prevents a test from preceding or succeeding itself.

After that, the following temporal requirements must be considered.

$$r_j + p_j \leq C_j \qquad \forall j \in J \tag{4.10}$$

$$d_j \geq C_j \qquad \forall j \in J \tag{4.11}$$

$$a_i + s_v + p_j - \bar{H}(2 - x_{i,j} - z_{v,i}) \leq C_j \qquad \forall i \in I, \forall j \in J, \forall v \in M_j \tag{4.12}$$

$$b_v + p_j - \bar{H}(2 - x_{i,j} - z_{v,i}) \leq C_j \qquad \forall i \in I, \forall j \in J, \forall v \in M_j \tag{4.13}$$

Constraint (4.10) ensures that the processing of test j starts only after the test has been released. Constraint (4.11) makes sure that the processing of test j is completed before the due date d_j. Once test j is assigned to prototype i belonging to variant v, it is necessary to make sure that it is processed only after prototype i becomes available and its initial set-up process is finished. This can be achieved by Constraint (4.12). Similarly, Constraint (4.13) represents the condition that test j cannot start until the maximum delay time of the components required for its corresponding prototype is reached.

Finally, we include the following additional constraints due to the problem-specific requirements.

$$C_j + p_k \leq C_k \quad \forall j \prec k, j, k \in J \tag{4.14}$$
$$x_{i,j} = x_{i,k} \quad \forall i \in I, \forall j \sim k,\ j, k \in J \tag{4.15}$$
$$x_{i,j} + x_{i,k} \leq 1 \quad \forall i \in I, \forall j \asymp k,\ j, k \in J \tag{4.16}$$
$$q_{j,k} = 1 \quad \forall j \in J \setminus J_{\text{Last}}, \forall k \in J_{\text{Last}} \tag{4.17}$$
$$x_{i,j} + x_{i,k} \leq 1 \quad \forall i \in I, \forall j, k \in J_{\text{Last}}, j \neq k \tag{4.18}$$

The precedence constraint can be formulated as shown in Constraint (4.14). For any pair of tests (j, k) with $j \sim k$, they must be assigned to the same machine and this is enforced by Constraint (4.15). Also, for tests (j, k) with $j \asymp k$, both tests must be processed on different prototypes, see Constraint (4.16). As the crash test must always be the last test executed, Constraint (4.17) restricts the crash test to follow other normal tests. Moreover, Constraint (4.18) does not allow two crash tests to be performed on the same prototype.

Notice that we use a large positive value to turn on or turn off enforcement of some constraints, for example, Constraint (4.7). However, this technique, known as Big-M formulation, usually leads to a poor lower bound as mentioned by Dyer and Wolsey [18].

4.2 Lower Bound Formulations

The first lower bound model is based on solving the set covering problem to satisfy the component requirements of all tests. The minimum number of prototype variants can be initially obtained. The energetic reasoning concept helps us further calculate the demand of each variant definitely required over the time intervals between the release and due dates.

Since the lower bound models still neglect how to sequence tests on each prototype, these solutions do not correspond to valid schedules. The result means that we need at least the same number of prototypes or more in order to carry out all tests. Nevertheless, the solution obtained from the simplified model can be regarded as the lower bound of the objective function to minimize the number of prototypes required.

4.2.1 Formulation Based on Set Covering

We notice that around 600 prototype variants exist in real-life instances. However in practice, it is not necessary to use all of them. It is sufficient to select a group of prototype variants from the given list such that they can satisfy the component requirements of all tests. Clearly, it is more convenient for the scheduling process to have variants which can be used for most of the tests, while special variants are selected only when necessary. Also, when the same variants are constructed repeatedly, the prototype workshop tends to have more experience which results in the reduction of the total construction time.

Lockledge et al. [40] previously showed that this selection problem is basically the same as the set covering problem. We want to select the smallest set of variants which still can perform all tests.

As one of the famous optimization problems, the set covering problem appears in several kinds of applications, for instance, crew scheduling. The set covering problem is NP-hard in the strong sense, see Garey and Johnson [20]. However, there are many efficient algorithms available as provided in the survey by Caprara and Toth [11]. Also, William [56] mentioned that the set covering problem is similar to the LP problem as the optimal solution must be one of the vertex points. However, the optimal vertex solution is not necessarily the same as the optimal solution obtained through solving its LP model.

To formally describe the problem, we define a binary variable for each variant v

$$u_v = \begin{cases} 1 & \text{if variant } v \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$$

Next, we introduce the MILP formulation as follows:

$$\text{minimize} \quad \sum_{v \in V} u_v \tag{4.19}$$

subject to

$$\sum_{v \in M_j} u_v \geq 1 \quad \forall j \in J \tag{4.20}$$

$$\sum_{v \in M_j \cap M_k} u_v \geq 1 \quad \forall j \sim k \,. \tag{4.21}$$

The objective function 4.19 minimizes the number of variants. For the basic set covering problem, we use Constraint (4.20) to guarantee that each test has at least one variant suitable for its component requirements. In addition, we can consider the constraints for two tests which must be assigned to the same prototype. Both tests require at least one common variant being able to perform them together. Thus, Constraint (4.21) is further included.

4.2.2 Formulation Based on Energetic Reasoning

The formulation based on set covering considers only the component requirements of all tests. It is important to include other characteristics of the problem like the temporal requirements. We further apply the idea of the relaxation model introduced by Hooker [22]. He suggested a model to minimize the number of late jobs according to the specified capacity of the cumulative resources. On the contrary, our problem aims to minimize the peak of resource usage such that no test is late. This concept is similar to energetic reasoning applied for constraint propagation in cumulative resources, see Baptiste et al. [4].

In this section, the concepts of set covering and energetic reasoning are combined in order to improve the lower bound. We first describe the fundamental knowledge of energetic reasoning before presenting the formulation.

4.2.2.1 Necessary Condition of Energetic Reasoning

First, we use parameters $t_1 \in \bar{R}$ and $t_2 \in \bar{D}$ with $t_1 < t_2$ to define the following sets:

- $J(t_1, t_2) = \{j \in J | t_1 \leq r_j, d_j \leq t_2\}$
- $J_l(t_1, t_2) = \{j \in J | r_j < t_1, t_1 < d_j \leq t_2,\, t_1 < r_j + p_j\}$
- $J_r(t_1, t_2) = \{j \in J | t_1 \leq r_j < t_2, t_2 < d_j,\, d_j - p_j < t_2\}$

- $J_{ol}(t_1, t_2) = \{j \in J | r_j < t_1, t_2 < d_j,\ 0 < r_j + p_j - t_1 \leq t_2 - d_j + p_j\}$
- $J_{or}(t_1, t_2) = \{j \in J | r_j < t_1, t_2 < d_j,\ r_j + p_j - t_1 > t_2 - d_j + p_j > 0\}$
- $J_{oa}(t_1, t_2) = \{j \in J | r_j < t_1, t_2 < d_j,\ r_j + p_j > t_2, d_j - p_j < t_1\}$

Figure 4.1 represents the concept of energetic reasoning. First, the necessary condition requires that we provide at least enough resources for processing all jobs in $J(t_1, t_2)$. We further include some parts of jobs which do not completely belong to $J(t_1, t_2)$, called the Left-Shift and Right-Shift conditions. J_l represents a set of jobs which can be left-shifted to start at their earliest start times r_j but cannot finish before t_1. Hence, the overlapping part of at least $r_j + p_j - t_1$ can be included. Similarly, for the right-shift condition, jobs in set $J_r(t_1, t_2)$ contribute at least $t_2 - d_j + p_j$. Also, we apply left-shifting or right-shifting to other jobs in order to determine the minimum contributing value ($\max\{0, \min\{r_j + p_j - t_1, t_2 - d_j + p_j, t_2 - t_1\}\}$). In this Left/Right-Shift category, tests in set $J_{ol}(t_1, t_2)$ and $J_{or}(t_1, t_2)$ contribute their minimum energy requirement when they are left- and right-shifted, while tests in $J_{oa}(t_1, t_2)$ consume the whole period between t_1 and t_2.

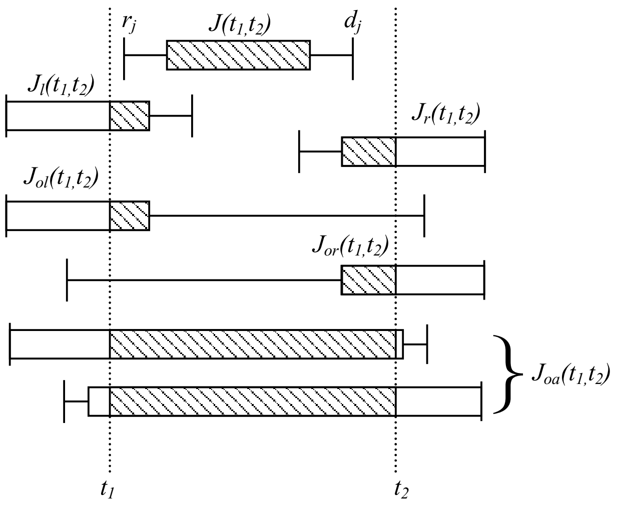

Figure 4.1: Representation of the energetic reasoning concept

4.2.2.2 MILP Formulation

In the formulation, we use the following decision variables.

- Variant demand variable

$$w_v \in \mathbf{N} = \{0, 1, \dots\} \quad \text{the number of required prototypes of variant } v$$

- Test-to-variant allocation variable

$$y_{v,j} = \begin{cases} 1 & \text{if test } j \text{ is assigned to a prototype of variant } v. \\ 0 & \text{otherwise} \end{cases}$$

- Prototype variant decision variable

$$z_{v,i} = \begin{cases} 1 & \text{if prototype } i \text{ belongs to variant } v \\ 0 & \text{otherwise} \end{cases}$$

The objective function (4.22) is to minimize the total number of required prototypes.

$$\text{minimize} \quad \sum_{v \in V} w_v \tag{4.22}$$

After that, we include the following constraints:

$$\begin{aligned} \sum_{v \in M_j} y_{v,j} &= 1 \quad \forall j \in J & (4.23) \\ w_v &\geq y_{v,j} \quad \forall j \in J, \forall v \in V & (4.24) \\ \sum_{v \in V} z_{v,i} &\leq 1 \quad \forall i \in I & (4.25) \\ \sum_{i \in I} z_{v,i} &= w_v \quad \forall v \in V. & (4.26) \end{aligned}$$

Constraint (4.23) assigns each test to exactly one variant which belongs to the eligible set of the test. Consequently, Constraint (4.24) guarantees that at least one prototype is built if the variant is selected for any test. Also, each prototype belongs to at most one variant, see Constraint (4.25). In addition, Constraint (4.26) balances the number of prototypes of each variant determined by variables $z_{v,i}$ and w_v.

The energetic reasoning conditions can be further included using

$$\begin{aligned} w_v(t_2 - t_1) \geq & \sum_{j \in J(t_1,t_2)} y_{v,j} p_j + \sum_{\substack{j \in J_l(t_1,t_2) \cup \\ J_{ol}(t_1,t_2)}} y_{v,j}(r_j + p_j - t_1) + \cdots \\ & + \sum_{\substack{j \in J_r(t_1,t_2) \cup \\ J_{or}(t_1,t_2)}} y_{v,j}(t_2 - d_j + p_j) + \sum_{j \in J_{oa}(t_1,t_2)} y_{v,j}(t_2 - t_1) + \cdots \\ & + \sum_{j \in J_{\text{Last}} | d_j < t_1} y_{v,j}(t_2 - t_1) + \sum_{\substack{i \in I | t_1 < a_i + s_v < t_2, \\ a_i + s_v \geq b_v}} z_{v,i}(a_i + s_v - t_1) + \cdots \\ & + \sum_{\substack{i \in I | t_1 < b_v < t_2, \\ a_i + s_v < b_v}} z_{v,i}(b_v - t_1) + \sum_{\substack{i \in I | a_i + s_v \geq t_2, \\ a_i + s_v \geq b_v}} z_{v,i}(t_2 - t_1) + \cdots \\ & + \sum_{\substack{i \in I | b_v \geq t_2, \\ a_i + s_v < b_v}} z_{v,i}(t_2 - t_1) \quad \forall v \in V, \forall j \in N_v, \forall t_1 \in \bar{R}, \forall t_2 \in \bar{D}. \end{aligned} \tag{4.27}$$

Constraint (4.27) considers the necessary energy consumption as explained in the previous section. As shown in Figure 4.2, we can additionally consider the impact of the crash test as if it still occupied the resource after its due date. Also, the total prototype construction time

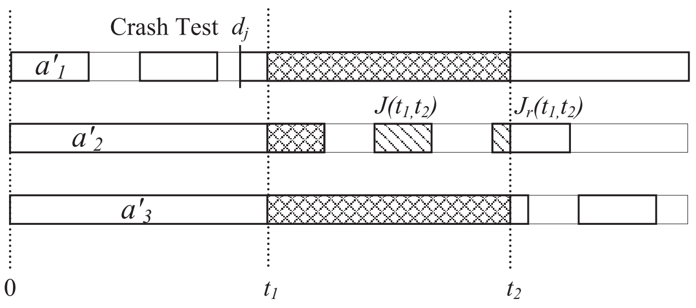

Figure 4.2: Impact of the crash test and the prototype construction time on the resource utilization

$a'_i = \max\{a_i + s_{v_i}, b_{v_i}\}$ can either partially overlap with the period considered or completely consume the entire gap if the prototype is not yet ready.

Furthermore, we apply the energetic reasoning concept for tests that must be executed on the same machine. Let us define $J_s(j) = \{j\} \cup \{k \in J | j \sim k\}$ as a set of tests that must be executed together with test j (including j itself). After that, we introduce the constraints:

$$t_2 - t_1 \geq \sum_{k \in J_s(j) \cap J(t_1,t_2)} y_{v,k} p_k \quad \forall v \in V, \forall j \in N_v, \forall t_1 \in \bar{R}, \forall t_2 \in \bar{D}, t_2 > t_1 \tag{4.28}$$

$$y_{v,j} = y_{v,k} \quad \forall j \sim k, \forall v \in V \tag{4.29}$$

Constraint (4.28) ensures that a resource with unit capacity can process all tests which must be allocated to the same prototype. The constraint is particularly useful since in our real-life instances sometimes more than two tests must be allocated together, for instance, when we have test j, k, and l where $j \sim k$ and $j \sim l$. Constraint (4.29) is further included since these tests certainly require the same prototype variant.

Finally, we can partially consider the effect of other problem specific constraints.

$$w_v \geq y_{v,j} + y_{v,k} \quad \forall j \asymp k, \forall v \in V \tag{4.30}$$

$$w_v \geq \sum_{j \in N_v \cap J_{\text{Last}}} y_{v,j} \quad \forall v \in V \tag{4.31}$$

For tests which must be done on different prototypes, if they use the same variant, Constraint (4.30) demands that at least two vehicles are available. Similarly, it is not possible to perform several crash tests on the same prototype. As in Constraint (4.31), the number of last jobs assigned to any variant must be less than the number of that prototype variant.

4.3 Computational Results

We solve the real-life data instances mentioned in Section 2.3. Our MILP solver is ILOG CPLEX 10.0 [24]. All computations in this report are performed on a Pentium Dual-Core, 3.0 GHz, and 4 GB RAM. We keep using CPLEX parameters at the default values.

As shown in Table 4.1, the complete MILP model can only be applied to solve small instance with 40 tests. No feasible solution can be found when the problem size increases to 100 tests. Our computer runs out of memory after solving the problem for about 36 hours. We obtain only a lower bound of 3.5 from the linear relaxation problem.

Notice that our objective function must always be an integer number. Actually, we can round up the lower bound to 4 prototypes. In fact, we can use the functionality in CPLEX to reject unnecessary incumbents. Alternatively, we simply include the constraint to ensure that the objective function must be greater than 4 before we repeat solving the complete MILP. After 5 days of computation, CPLEX can neither find a solution nor further improve the previous lower bound.

As the size of the MILP formulation explodes rapidly for other larger instances, we cannot even start the solving process. Our computer runs out of memory before completely translating the model to a machine-readable format.

Note that regardless of the amount of memory available in a computer, CPLEX always has the memory restriction of 2GB when running on a 32-bit computer platform [28]. To access more memory, it is necessary to use a 64-bit machine platform.

Table 4.1: Minimizing the number of required prototypes by solving the complete MILP model

n	#Variable	#Constraint	m_r	Total time (s)
41	2,828	78,298	5^a	233.86
100	13,602	92,393	3.5^b	132,284.36
231	72,072	8,465,220	-	-
487	425,106	134,373,138	-	-

Next, we apply the set covering and energetic reasoning formulations to realize the lower bound of the number of required prototypes. The computational results are shown in Table 4.2. All instances can be optimally solved within a relatively short time.

As the solutions obtained from the set covering model refer to the minimum number of required prototype variants, they can initially provide us with a lower bound of the objective function for those large instances. However, these lower bounds are rather poor since only the component requirements are included.

With the help of energetic reasoning, the lower bounds are further improved since we also can consider the effect of other constraints like release dates, due dates and prototype availability times.

Note further that the size of an MILP model is always critical for solving a large problem. We can eliminate some unnecessary variables in $y_{v,j}$ as the relations between some tests and variants are irrelevant due to the component requirements. Thus, the size of the master MILP is substantially reduced when the tests are restricted to fewer variants.

Moreover, the size of our MILP model depends on the number of distinct values of release and due dates. If each job always has its own values, the model can become too large to be solved. This situation may happen more often in case of a randomly generated instance, while in practice these values tend to be assigned in a duplicate fashion. As shown in Table 2.1, even the instance of 487 tests totally contains around 80 values of distinct release and due dates. While it is necessary to have a simplified model that is as accurate as possible, we should also keep in mind that the size of the formulated MILP problem tends to explode in large scale problems.

Table 4.2: Determining the lower bound of the number of required prototype

n	#Variable	#Constraint	$\bar{m}_r$	Total time (s)
Solve the set covering model				
41	38	42	5	0.04
100	3	101	2	0.02
231	9	234	9	0.03
487	663	521	101	0.84
Solve the energetic reasonning model				
41	1,353	24,957	5	0.69
100	456	183	4	0.23
231	1,482	8,047	18	0.26
487	111,545	497,998	110	69.03

In practice, the heuristic or meta-heuristic approaches are usually implemented to solve scheduling problems. The role of MILP is quite substantial. Mostly people formulate and solve their MILP models for comparison with their approaches only in case of small size instances. The results from this analysis tend to be extrapolated for real larger instances.

We show that when appropriate, the structure of the problem at hand can help us to formulate the simplified MILP such that the true lower bound can be obtained even from large data instances for further optimality gap measurement.

Chapter 5

CP Approach

In this section, we apply Constraint Programming to solve our scheduling problem. First, we need to formulate the CP model whose objective function is to minimize the number of required prototypes. Next, we suggest our specific search strategy to further improve the performance of search algorithms.

After that, we briefly provide a technical introduction to the CP solvers we used in this thesis, before comparing both the solutions and performance of our search strategy with the default search available in our CP solvers.

Finally, we provide trade-off analyses to consider the impact of the increase and decrease in the number of available prototypes on the completion time and the number of tests executed on time.

5.1 CP Formulation

It is quite common in the CP community to define a resource constraint in a general scheduling problem by the term **cumulative**$(\vec{t}, \vec{p}, \vec{c}, C)$, where $\vec{t} = [t_1, .., t_n]$, $\vec{p} = [p_1, .., p_n]$, $\vec{c} = [c_1, .., c_n]$ are the vectors of starting times, processing times, consumption rates of jobs, respectively, and C is the capacity of the machine. The constraint is satisfied if the condition $\sum_{j \in J_t} c_j \leq C$ holds for all time instances t in the valid time frame, where $J_t = \{j \in J | t_j \leq t \leq t_j + p_j\}$ is the set of tasks that are in progress at time t. Therefore, the total consumption of all jobs $j \in J_t$ cannot exceed capacity C.

In our problem, each prototype i is considered as a resource with capacity $C = 1$ since it can perform at most one test at a time. Also, the consumption rate of test j is $c_j = 1 \; \forall j \in J$ as each test requires a single prototype.

Next, we define the following variables:

- $v_i \in V \; \forall i \in I$ represents the variant of prototype i
- $x_j \in I \; \forall j \in J$ represents the prototype which is selected to carry out test j
- t_j represents the start time of test j

To give an example for variable x_j, we assume that set I and set J contain 5 prototypes and 5 tests. When the sequence of the allocation of each test is $\vec{x} = [x_1, x_2, x_3, x_4, x_5] = [1, 2, 2, 3, 1]$, it means that we assign all tests to just three prototypes. The first prototype ($i = 1$) executes test 1 and test 5. The second prototype ($i = 2$) executes test 2 and test 3.

The third prototype ($i = 3$) executes test 4. Also, the number of required prototypes can be determined from the maximum value in the sequence.

The CP formulation is presented as follows:

$$\text{Minimize} \quad \max_{j \in J} x_j \tag{5.1}$$

subject to

$$t_j \geq r_j \quad \forall j \in J \tag{5.2}$$

$$t_j + p_j \leq d_j \quad \forall j \in J \tag{5.3}$$

$$\text{cumulative}(t_j | x_j = i, p_j | x_j = i, c_j = 1 | x_j = i, 1) \quad \forall i \in I \tag{5.4}$$

$$v_i \notin M_j \Rightarrow x_j \neq i \quad \forall i \in I,\ \forall j \in J \tag{5.5}$$

$$x_j = i \Rightarrow t_j \geq a_i + s_{v_i} \quad \forall i \in I,\ \forall j \in J \tag{5.6}$$

$$x_j = i \Rightarrow t_j \geq b_{v_i} \quad \forall i \in I,\ \forall j \in J \tag{5.7}$$

$$t_j + p_j \leq t_k \quad \forall j \prec k,\ j, k \in J \tag{5.8}$$

$$x_j = x_k \quad \forall j \sim k,\ j, k \in J \tag{5.9}$$

$$x_j \neq x_k \quad \forall j \asymp k,\ j, k \in J \tag{5.10}$$

$$x_j = x_k \Rightarrow \ t_j \geq t_k + p_k \quad \forall j \in J_{\text{Last}}, \forall k \in J, j \neq k \ . \tag{5.11}$$

The objective function (5.1) minimizes the maximum number of prototypes used to allocate all tests. Constraint (5.2) and Constraint (5.3) state that tests must be performed between their release and due dates. Constraint (5.4) is a resource constraint with the term $t_j | x_j = i$ denoting the tuple of start times for tests that are assigned to prototype i, i.e. $x_j = i$. According to the definition of the cumulative constraint, we prohibit the concurrent execution of two jobs which are assigned to the same machine. Constraint (5.5) prevents that a test is assigned to a prototype whose variant does not belong to the eligibility set of the test.

Constraint (5.6) and Constraint (5.7) ensure that a test on machine i cannot start before the availability of the prototype according to our model. Notice that both set-up time s_{v_i} and the availability time of components b_{v_i} depend on the value of variable v_i, which is not known a priori. It is another benefit of the CP approach that variables can index parameter arrays.

Constraint (5.8) represents the precedence constraints. Constraint (5.9) ensures that any pair of tests (j, k) with $j \sim k$ will be executed on the same machine. Also, tests j and k with $j \asymp k$ must be processed on different machines as restricted by Constraint (5.10). Finally, Constraint (5.11) guarantees that test $j \in J_{Last}$ is the last job executed on the machine to which it is allocated.

5.2 Search Strategy

Our search strategy tries to obtain a good feasible solution as soon as possible rather than to find an optimal one. We separate the branching scheme into three parts with respect to our decision variables: x_j, v_i, and t_j.

1. Our strategy selects the most critical test j first. Among all prototypes that are able to carry out this test, we try to select the one which has the lowest index i, as our objective is to minimize $\max_{j \in J} x_j$ which represents the number of prototypes required.

To measure the priority of each test, we consider the number of remaining possible prototypes as the first criterion. A tie is further broken by either the minimum latest start time (lst_j) or the minimum slack time ($lst_j - est_j$).

Initially, the domain of x_j or a range of prototypes possible to execute test j is $[1, ..., m]$. That means we can assign test j to any prototype. As prototype 1 has the lowest index, it is selected to execute test j, i.e. $x_j = 1$. Due to the component requirements of test j, we can realize which tests are no longer possible to be executed on prototype 1. The domains of these tests must be reduced to $[2, ..., m]$. The mechanism of constraint propagation helps us consider also the effect of other temporal and additional constraints.

The domain of x_j further shrinks during the search. Therefore, a test with a lower degree of freedom (fewer allocatable prototypes) receives a higher priority as it becomes more difficult to find an appropriate prototype. The slack time and latest start time are used to measure the criticalness in the time domain. A test with a small slack time tends to be more difficult to schedule. Also, due to the construction time of prototypes, the number of prototypes available at the beginning is limited. Thus, we give higher priority to a test with a smaller latest start time.

2. We simply instantiate the variant v_i of prototype i with any possible value in its remaining domain. Notice that at the root node the domain of v_i starts with a complete range in domain V. This means prototypes can be assigned to any variant. After the test allocation process, the domain automatically shrinks as the constraint propagation eliminates variants which cannot satisfy the component requirements of the assigned tests.

3. The start time of each test is determined by using the schedule-or-postpone method, see Le Pape et al. [35]. The algorithm selects an unscheduled job with minimum earliest start time and tries to schedule it as early as possible. However, during backtracking the job is postponed. It means that the algorithm can neglect this job until its domain of earliest start time is increased due to the constraint propagation mechanism.

The complete branching scheme is shown in Algorithm 1. For the rest of this thesis, the notations LST/MinId and Slack/MinId refer to these search schemes using latest start time and slack time for the job selection criteria, while trying to choose the machine with the minimum index to perform the selected job. The depth-first search is applied as we want to achieve a feasible solution as soon as possible.

5.3 CP Solvers

To solve the CP problem, we apply ILOG OPL 3.7 [25] and ILOG Scheduler 6.2 [26]. OPL is a programming language for representing optimization problems. As shown in Figure 5.1, OPL is built on top of optimization engines: ILOG CPLEX [24] for linear and integer programming; ILOG Solver [27] for constraint programming; and ILOG Scheduler [26] for constraint-based scheduling. Therefore, one problem can be formulated and solved using different approaches. Also, OPL provides a script language allowing users to iteratively solve optimization problems or implement hybrid strategies.

Algorithm 1 Branching scheme for test scheduling: LST/MinId and Slack/MinId

```
%PART 1) Assign tests to prototypes
%Let $J_a$ be the set of tests not yet allocated
$J_a := J$
**while** $J_a \neq \emptyset$ **do**
  - Select test $j$ using criteria:
    1) smallest number of possible machines
    2) minimum latest start time or slack time
  - Select a possible machine $i$ having the lowest index $i$
  - Try
    $x_j = i$
   or
    $x_j \neq i$ (for backtracking)
  - $J_a = J \setminus \{j\}$
**end while**

%PART 2) Assign prototype variants
**while** not all prototypes have specified variants **do**
  - Select prototype $i$ arbitrarily
  - Try to instantiate $v_i$ with possible values in its remaining domain
**end while**

%PART 3) Assign start times of tests (SetTimes)
%Let $J_s$ be the set of tests not yet postponed
$J_s := J$
**while** not all tests have fixed start times **do**
  **if** $J_s \neq \emptyset$ **then**
    - Select test $j$ from $J_s$ using criteria:
      1) minimum earliest start time
      2) minimum latest end time
    - Try
     $t_j$ equals its earliest start time
      or
     postpone test $j$ until its domain has been changed
  **else**
    Backtrack to the most recent choice point
  **end if**
  Update $J_s$:
   - remove tests which are postponed
   - include tests whose domains are changed
**end while**
```

Moreover, to solve CP problems on OPL, users can simply create a model without suggesting a search. A specific search can also be implemented in order to improve the performance. Compared to ILOG Solver, OPL provides high level modeling functionalities such that a complex search procedure can be written in a concise and natural way, see Van Hentenryck et al. [55].

Also, we apply ILOG Scheduler to solve our problem. Extended from ILOG Solver, ILOG Scheduler provides the functionalities to implement constraint-based scheduling problems. Unlike OPL, users have to develop a model using the standard programming languages like C++ or Java. Although ILOG Scheduler provides some basic search algorithms, users must specifically implement a search together with the model. This certainly requires more effort and experience. Nevertheless, ILOG Solver and Scheduler are inevitable for developing advanced strategies. For instance, users can implement their node selection policy instead of relying on the standard ones like depth-first search.

To apply the search algorithms available in ILOG Scheduler, we replace the resource allocation process in our search Algorithm 1 with the algorithm called AssignAlternative, while the rest of the algorithm still remains the same. In case of parallel machines with unit capacity, we can try the AssignAlternative algorithm with three possible options: SelAltRes, SelResMinGlobalSlack, and SelResMinLocalSlack. Moreover, the schedule-or-postpone method used in the third part of Algorithm 1 is available as the SetTimes algorithm.

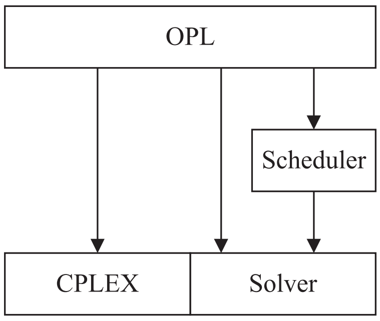

Figure 5.1: Architecture of ILOG optimization software.

5.4 Computational Results

We formulate the CP model presented in Section 5.1 using ILOG OPL 3.7 [25] and ILOG Scheduler 6.2 [26]. As mentioned in Section 5.3, we solve the problem using the search methods available in OPL and Scheduler before comparing them with our search strategy in Section 5.2. The time limit is set at 5 hours for the largest instance and 1 hour for the rest.

In ILOG Scheduler, we need to select appropriate constraint propagation rules apart from implementing the model and search. There are several standard rules available in ILOG Scheduler. First, we tried the lowest propagation strength containing only the light precedence graph and disjunctive constraints. But with this approach we cannot get any solution even for the smallest instance. Therefore, the edge-finding technique must be applied. Also, we tried to use the strongest propagation level containing the balance constraint. However, the computation cost is so high that our computer (Pentium Dual-Core, 3.0 GHz, 4 GB RAM) runs out of memory when solving the largest data instance. Hence, we did not use the balance constraint for all computations.

Regarding the range of available machines, we might start by setting $m = n$. That means in the worst case it is possible that each test has its own prototype for execution. But as in practice many tests can share the same prototype, it suffices to specify $m = \lceil n/3 \rceil$. Notice that, in some cases we may not obtain a feasible solution even if such a large number of machines is already specified. For instance, prototypes may be available too late if there are many tests with very early due dates. However, it suggests that the conflicts must arise from the temporal constraints rather than from the resource constraints.

The computational results in Table 5.1 show the comparison between the minimum number of required prototypes obtained from several search algorithms. As we use three different search algorithms in ILOG Scheduler, we select only the best result of them to present here. The detailed results are shown in Table A.1. The number of variables and constraints corresponds to the CP model formulated in OPL. Also, the Gantt chart for the case of 100 tests is provided in Figure 5.2, where test 0 represents the total construction time of each prototype.

For these instances, our algorithms outperform the standard search approaches in OPL and Scheduler because these standard searches are generally designed to balance the load in the set of available resources rather than to minimize the peak resource demand. Note that CP can prove the optimality of its solution only for the smallest instance. For the other cases it manages to achieve feasible solutions within a few minutes.

Note that in general provided there is enough time, the solver either achieves an optimal solution for this number of prototypes or proves that the problem is infeasible. Due to limitations of computation time in practice, the solver may not be able to prove the infeasibility of the problem. Even if it finds a feasible solution we cannot be sure about its optimality as long as there still are unexplored nodes in the search tree.

Table 5.1: Minimizing the number of required prototypes by CP.

n	#Var	#Const	OPL		Scheduler		LST/MinId		Slack/MinId	
			m_r	Time(s)	m_r	Time(s)	m_r	Time(s)	m_r	Time(s)
41	793	2,120	5[a]	0.98[b]	5[a]	0.05[c]	5[a]	0.04[b]	5[a]	0.04[b]
100	3,935	10,636	5	1.11	5	0.51	5	0.29	5	0.29
231	19,253	70,019	19	14.82	23	6.42	18	8.15	18	6.42
487	76,088	331,957	111	242.03	121	1,398.36	111	131.73	111	85.79

[a] Optimal solution
[b] Total computation time
Time: Computation time to achieve the solution

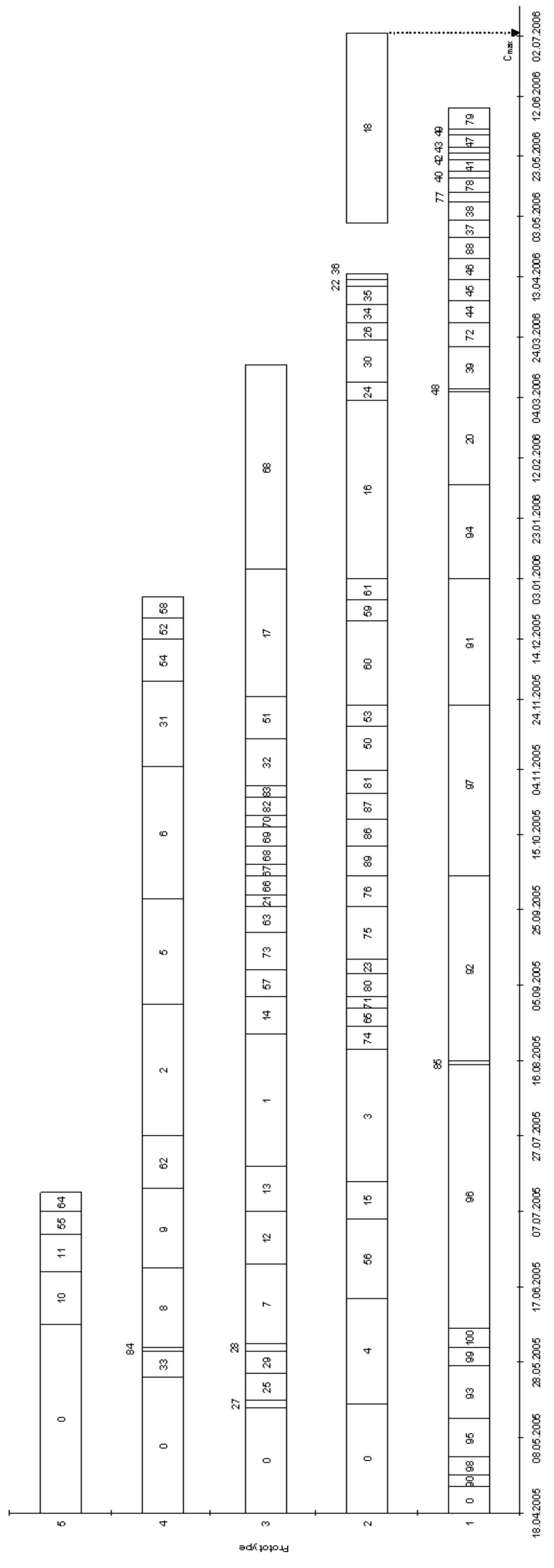

Figure 5.2: Gantt chart of the instance with 100 tests

Furthermore, we apply CP to further determine $\bar{m}_r$, the lower bound of the number of required prototypes. First, notice that our search scheme Algorithm 1 tries to achieve a good feasible solution as soon as possible. To focus on tightening the lower bound, we may apply another search scheme and strategy to always consider the best node, which leads to the lowest cost function, in the search tree.

Alternatively, we apply the concept of destructive proof by simply checking the feasibility of the CP model, that is, the objective function (5.1) is neglected in order to find a feasible solution only. The procedure shown in Algorithm 2 starts by setting one prototype available and further increments the number of prototypes as long as the solving problem is found infeasible. We consider the final value as the lower bound which can be obtained from using CP approach.

Algorithm 2 Destructive procedure to determine the lower bound

```
$m = 1$
**repeat**
    Solve the feasibility of the CP model
    $m \Leftarrow m + 1$
**until** Problem is feasible OR cannot be solved within the time limit
```

We apply Algorithm 2 to all instances except the instance of 41 tests where the optimal solution has already been obtained. The computational results are shown in Table 5.2. For other large instances, CP can only prove that the solving problem is infeasible within the limit time if the number of prototypes is small enough. The instances with 100, 231, and 487 tests have their lower bounds at 3, 9, and 9, respectively. Remember that our best feasible solutions of these instances require 5, 18, and 111 prototypes, respectively. Therefore, we encounter a large set of numbers of prototypes for which our CP approach fails to prove feasibility or infeasibility within a reasonable time. We call this set *the problem gap*. In the case of the largest instance, the problem gap between 10 to 110 prototypes is obviously too large.

Moreover, we provide the comparison between our CP solutions and the lower bounds obtained from the simplified MILP model in Table 5.3. Previously, we could optimally solve the instance of 41 tests by both MILP and CP. With the lower bound of the simplified MILP model, we can realize that the solution obtained from CP for the instance of 231 tests is also optimal.

In addition, the remaining gaps of other instances are just 1 prototype. Therefore, CP can handle a large instance and find good feasible solutions. Also in practice, such a small gap might be negligible as the manufacturer may prefer to have a more flexible schedule, rather than the very tight and optimal one.

Table 5.2: Determining the lower bound of the number of required prototypes using CP.

n	m	#Var	#Const	Total computation time(s)			
				OPL	Scheduler	LST/MinId	Slack/MinId
100	1	409	507	0.00	0.01	0.01	0.01
	2	515	613	0.05	0.04	0.01	0.01
	3	621	719	0.12	-	0.01	0.02
231	1	933	1,858	0.04	0.02	0.01	0.02
	2	1,170	2,789	0.10	0.03	0.03	0.03
	3	1,407	3,720	0.18	0.04	0.04	0.06
	4	1,644	4,651	0.42	0.09	0.11	0.13
	5	1,881	5,582	1.80	0.48	0.30	0.67
	6	2,118	6,513	11.96	15.61	1.54	8.97
	7	2,355	7,444	119.68	1,003.07	10.39	111.92
	8	2,592	8,375	1,152.15	-	78.58	1,543.87
	9	2,829	9,306	-	-	766.20	-
487	1	1,984	2,961	0.18	0.09	0.08	0.09
	2	2,477	3,973	0.30	0.20	0.19	0.19
	3	2,970	4,985	0.58	0.36	0.33	0.32
	4	3,463	5,997	1.52	1.48	0.78	0.68
	5	3,956	7,009	6.17	6.18	2.40	2.20
	6	4,449	8,021	39.15	30.44	9.71	10.07
	7	4,942	9,033	283.75	181.00	49.63	57.02
	8	5,435	10,045	2,338.19	1,226.72	308.76	330.20
	9	5,928	11,057	17,625.10	9,325.13	2,385.11	2,378.18

-: No result within time limit

Table 5.3: Comparison between the number of required prototypes and the lower bounds obtained from CP and MILP.

n	m_r	Lower Bound ($\bar{m}_r$)		Optimality Gap
		CP	Simplified MILP	
41	5[a]	5	5	0
100	5[c]	4	4	1
231	18[b]	9	18	0
487	111[c]	9	110	1

[a] Optimal solution proved by CP
[b] Optimal solution proved by MILP
[c] Feasible solution

5.5 Trade-off Analysis Between Makespan and Number of Prototypes

Although our main objective is to reduce the number of required prototypes, it is interesting to analyze how the number of prototypes can affect the total completion time. The manufacturer might prefer to build more prototypes in order to gain a shorter testing period and start mass production earlier.

To perform the trade-off analysis, we apply the classical approach of parameterization, that is, we fix the number of prototypes and minimize the makespan with respect to all constraints since in this problem the number of prototypes is discrete, while the makespan is (almost) continuous. We slightly modify the CP model described in Section 5.1 by changing the objective function to minimize the makespan $C_{\max}$ and by including a constraint to ensure that the makespan is not smaller than the completion time of any test. The resulting model called *CP-Cmax* model is provided as follows:

$$\begin{aligned}
&\text{Minimize} \quad C_{\max} \\
&\text{subject to} \\
&\quad \text{Constraint (5.2)–(5.11)} \\
&\quad C_{\max} \geq t_j + p_j \qquad\qquad \forall j \in J. && (5.12)
\end{aligned}$$

Note that this optimization problem can be represented by $P_m|r_j, d_j, M_j, prec|C_{max}$ which is NP-hard in the strong sense as a generalization of $P_m|prec|C_{max}$, see Pinedo [48].

We suggest the trade-off procedure in Algorithm 3. Initially, we assume the availability of a large number of prototypes such that the makespan only depends on the temporal constraints. Like in the previous section, we may define $m = \lceil n/3 \rceil$.

After solving the problem, we determine m_r, the number of prototypes actually required from the solution. Notice that although our new objective is to minimize the makespan, the algorithm may find a solution using a number of prototypes lower than the one initially specified. Also, we reduce the number of available machines from m_r to further realize how much the makespan will be changed. That means we skip computations between m and m_r. This procedure is repeated until the CP-Cmax model becomes infeasible or the calculation cannot terminate within a given time limit.

Algorithm 3 Trade-off procedure between the makespan and the number of prototypes

```
$m \Leftarrow \lceil n/3 \rceil$
**repeat**
  Solve the CP-Cmax model
  Determine $m_r$, the number of required prototypes, from the obtained feasible solution
  $m \Leftarrow m_r - 1$
**until** Problem is infeasible OR cannot be solved within the time limit
```

The computational results of the trade-off analyses are provided in Table 5.4. First, we use OPL with default search as the CP solver. We then obtain the makespan and the number of required prototypes according to the specified number of prototypes. Additionally, we provide the corresponding results computed by search algorithms in Scheduler and our search algorithms: LST/MinId and Slack/MinId. For Scheduler we select only the best results to present here. The detailed computations of the instances with 231 and 487 tests are available in Table A.2, A.3, and A.4

In the case of the instance with 41 tests, we start from $m = 14$ and get a solution using 5 prototypes with the optimal makespan of 330 days. However, if we set $m = 4$ the problem becomes infeasible as it has not enough prototypes to execute all tests with respect to all constraints. Also, for other larger instances we can achieve optimal solutions at the first iteration ($m = \lceil n/3 \rceil$). But after reducing a number of vehicles, we only get feasible solutions with a slight increase in the makespan.

For these large instances, CP mostly obtains the feasible solutions. The optimal solutions are found only when the given number of prototypes is large enough. As the resource capacity is so high, our scheduling problem is mainly restricted by the temporal constraints. It is similar to solving $P_m||C_{\max}$ which normally is NP-hard in the strong sense. However, an efficient polynomial time algorithm exists when $m \geq n$, see Pinedo [48].

We notice that during the trade-off procedure, the number of variables and constraints becomes smaller in each iteration. However, the slight decrease in the problem size alone cannot compensate for the complexity caused by the resource constraints.

Moreover, it should be noted that an optimal value of the makespan in one iteration can be regarded as a lower bound of the makespan in the next iteration. This may reduce the search space as the constraints are tighter. However, this idea does not work effectively here since in each case where an optimal solution is found the makespan is actually determined by the earliest completion time ($r_j + p_j$) of a test with a very long processing time. Normally during constraint propagation, a lower bound of the makespan is already constrained to be greater than or equal to the earliest completion times of all jobs. Therefore, it does not help CP when we try to include the same condition.

Finally, we illustrate the trade-off analysis in case of the instance with 487 tests in Figure 5.3. Although our search strategies aim to minimize the number of required prototypes, they can achieve solutions with a shorter makespan compared to OPL and Scheduler. The analysis also shows that the manufacturer can reduce the makespan by roughly 100 days (or from 501 to 398 days) if the cost of 10 additional prototypes is acceptable.

Table 5.4: Minimizing the makespan with a given number of prototypes.

n	m	#Var	#Const	OPL			Scheduler			LST/MinId			Slack/MinId		
				m_r	$C_{\max}$	Time (s)	m_r	$C_{\max}$	Time (s)	m_r	$C_{\max}$	Time (s)	m_r	$C_{\max}$	Time (s)
41	14	784	1,383	5	330[a]	0.33[b]	6	330[a]	0.07[b]	5	330[a]	0.11[b]	5	330[a]	0.10[b]
	4	314	513	Infeasible		0.06[b]	Infeasible		0.02[b]	Infeasible		0.03[b]	Infeasible		0.01[b]
100	34	3,907	4,005	5	312[a]	0.90[b]	7	312[a]	0.29[b]	5	312[a]	0.34[b]	5	312[a]	0.56[b]
	4	727	825	-	-	-	-	-	-	-	-	-	-	-	-
231	77	18,945	72,614	21	376[a]	27.02	24	376[a]	5.82	21	376[a]	5.20	21	376[a]	5.24
	20	5,436	19,547	20	376[a]	16.25	-	-	-	20	376[a]	3.35	20	379	3.10
	19	5,199	18,616	19	379	13.74	-	-	-	19	379	3.07	19	381	2.90
	18	4,962	17,685	18	381	10.18	-	-	-	18	381	2.65	-	-	-
	17	4,725	16,754	-	-	-	-	-	-	-	-	-	-	-	-
487	162	81,357	165,893	128	381[a]	27,860.45[b]	162	381[a]	6,392.91[b]	124	381[a]	6,763.01[b]	124	381[a]	8,960.50[b]
	127	64,102	130,433	127	393	6,700.25	127	445	2,134.55	no computation			no computation		
	126	63,609	129,461	126	394	15,122.48	126	445	2,082.01	no computation			no computation		
	125	63,116	128,449	125	401	3,247.89	no computation			no computation			no computation		
	124	62,623	127,437	124	404	2,221.32	124	500	1,343.50	no computation			no computation		
	123	62,130	126,425	123	413	824.28	123	504	1,265.69	123	389	1,571.32	123	388	1,989.30
	122	61,637	125,413	122	414	773.89	122	515	1,203.77	122	401	625.42	121	397	1,203.47
	121	61,144	124,401	121	415	738.71	121	526	1,137.86	121	401	596.77	121	398	1,087.50
	120	60,651	123,389	120	435	653.08	120	535	1,043.81	120	413	276.38	120	399	1,174.37
	119	60,158	122,377	119	438	619.52	-	-	-	119	413	263.62	119	430	191.19
	118	59,665	121,365	118	481	463.57	-	-	-	118	433	231.50	118	452	160.36
	117	59,172	120,353	117	483	436.39	-	-	-	116	443	168.25	116	453	149.47
	116	58,679	119,341	116	483	485.32	-	-	-	116	443	160.05	116	453	142.12
	115	58,186	118,329	115	494	412.39	-	-	-	115	453	145.94	115	478	14,530.80
	114	57,693	117,317	114	498	373.08	-	-	-	114	481	129.24	114	486	113.42
	113	57,200	116,305	113	501	342.57	-	-	-	113	489	119.70	113	486	108.65
	112	56,707	115,293	112	517	284.13	-	-	-	112	498	106.77	112	498	91.04
	111	56,214	114,281	-	-	-	-	-	-	111	501	93.27	111	501	84.93

[a] Optimal solution [b] Total computation time Time: Time to achieve the solutions -: No result within the time limit

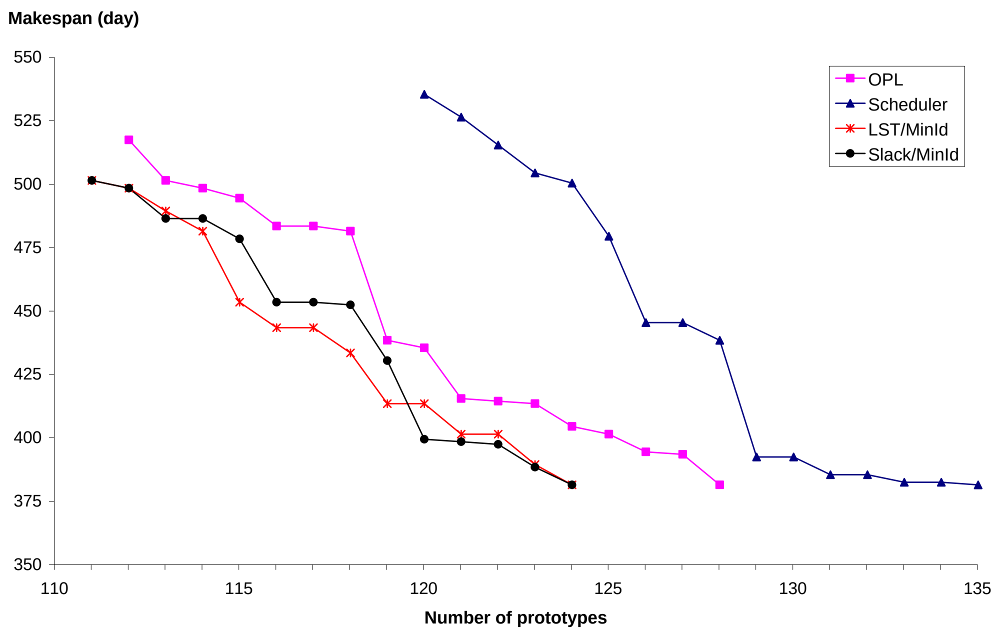

Figure 5.3: Trade-off between the makespan and the number of prototypes for the instance with 487 tests.

5.6 Trade-off Analysis Between the Number of Executed Tests and Prototypes

In contrast to the previous section, we here assume that the manufacturer tries to use less prototypes than required. As not all tests can be executed within their due dates, we try to maximize the number of tests which can be completed on time.

First, we modify our CP model in Section 5.1 to consider the total number of late jobs as the new objective function, while the due date constraint is neglected. We further introduce Constraint (5.13) to ensure that binary variable $U_j = 1$ if test j is late; otherwise zero. The *CP-Late* model is then provided as follows:

$$\begin{aligned} &\text{Minimize} \quad \sum_{j \in J} U_j \\ &\text{subject to} \\ &\quad \text{Constraint (5.2), (5.4)–(5.11)} \\ &\quad C_j > d_j \Rightarrow U_j = 1 \qquad\qquad \forall j \in J. \qquad (5.13) \end{aligned}$$

Note that this optimization problem can be represented by $P_m|r_j, d_j, M_j, prec|\sum_{j\in J} U_j$ which is NP-hard in the strong sense as a generalization of $1|r_j|\sum U_j$, see Pinedo [48].

However, we find that the performance of CP becomes very poor. As shown in Table 5.5, we solve the CP-Late model using various search algorithms. We specify the number of prototypes with sufficient values which are determined from our feasible solutions. However, the solutions obtained from the CP-Late model contain many late jobs instead of none.

In this case, after relaxing the due date constraint, the scheduling problem becomes quite loose. Constraint propagation cannot efficiently tighten the domain of variables. Also, our search strategies are not appropriate for this objective function since it keeps trying to unreasonably allocate more tests on the lowest index prototype. We can determine which tests are late only after their start times are assigned at the end of our search. CP can just realize that the final result leads to so many late jobs. The effort has already been taken in vain to search in this region.

Table 5.5: Minimizing the number of late jobs using CP.

n	m	number of late jobs			
		OPL	Scheduler	Slack/MinId	LST/MinId
41	5	8	5	0	8
100	5	90	89	90	90
231	19	124	-	-	-
487	111	128	128	101	101

-: No result within time limit

To properly deal with the problem $P_m||\sum U_j$, it becomes necessary to implement another specific search strategy. We cannot separately consider the allocation and sequence parts. For each job, both decisions must be more dynamically interleaved.

Instead of solving the complete problem directly, we apply the procedure to iteratively determine the variant and the test allocation for each prototype. Thus, we can apply CP to solve just a single machine problem to minimize the number of late tests. Although we may lose a globally optimal solution, this strategy allows us to solve smaller problems more efficiently by standard approaches.

However, the precedence constraints cannot properly be included in this procedure since we solve the problem iteratively. Thus, we apply the technique used in the heuristic algorithm developed by Scheffermann et al. [53]. As they noticed that only few tests require this requirement and the corresponding release and due dates are quite close, it is acceptable to adjust the due date of the earlier test to be the release date of the following test.

Let us define $J_s \subseteq J$ as a set of tests which are considered for scheduling in each iteration. Also, $J_m \subseteq J_s$ is a set of the tests which must be scheduled within their due dates. Note that we use set J_m to give higher priority to some tests which are definitely assigned to this prototype.

We formulate the *CP-Late-Single* model as follows:

$$\text{Minimize} \quad \sum_{j \in J_s} U_j$$

subject to

$$t_j \geq r_j \qquad \forall j \in J_s \tag{5.14}$$

$$C_j > d_j \Rightarrow U_j = 1 \qquad \forall j \in J_s \tag{5.15}$$

$$C_j \leq d_j \qquad \forall j \in J_m \tag{5.16}$$

$$\text{cumulative}(t_j, p_j, c_j, 1) \qquad \forall j \in J_s \tag{5.17}$$

$$t_j \geq a_i + s_{v_i} \qquad \forall j \in J_s \tag{5.18}$$

$$U_j = U_k \qquad \forall j \sim k,\ j,k \in J_s \tag{5.19}$$

$$U_j + U_k \geq 1 \qquad \forall j \asymp k,\ j,k \in J \tag{5.20}$$

$$U_j = 0 \wedge U_k = 0 \Rightarrow t_j \geq t_k + p_k \quad \forall j \in J_{\text{Last}} \cap J_s, \forall k \in J_s, j \neq k\ . \tag{5.21}$$

Notice that this model considers only tests provided in J_s. Also, our procedure will actually select tests with $U_j = 0$ to be executed on this prototype. We will try to schedule other late tests again on the next possible prototypes.

Constraint (5.15) is applied for any test in J_s when test j is late, its variable U_j becoming one. Only tests $j \in J_m$ must be on schedule as restricted by Constraint (5.16). Constraint (5.17) is a resource constraint of a single machine.

Constraint (5.19) ensures that test j and k with $j \sim k$ must be assigned to the same prototype. It means either both tests are performed within their due dates on this machine or will be considered for scheduling later on the next prototypes. Also, for test j and k with $j \asymp k$, they must be processed on different machines due to Constraint (5.20). Finally, Constraint (5.21) ensures that tests $j \in J_{\text{Last}}$ will be executed in the last sequence if it is assigned to this machine.

Our procedure is shown in Algorithm 4. We define J_a as a set of tests not yet allocated. Also, J_n represents a set of tests neglected because they certainly cannot be completed on schedule.

In the initial stage, we apply the set covering MILP model in Section 4.2.1 to determine the minimum set of required variants V_r. Thus, we need not consider such a large number of all variants. After that, the main procedure starts from allocating tests on the first prototype $i = 1$ and continues to use more prototypes until there are no more tests left in J_a.

Inside the loop, we first check whether it is still possible to execute tests within their due dates. If the availability time of a prototype plus the minimum set-up time is larger than the latest start time $(d_j - p_j)$ of any test, then the test is put into J_n.

After that, for each prototype variant we solve the CP-Late-Single model twice. First, we specify J_m to be an empty set and J_s to be a set in which tests can be performed only by this variant. That means we try to schedule these tests first because we want to avoid building the same prototype variant just for them. Second, we consider all other tests which are also possible to be allocated to this variant. These tests receive lower priority because they still have chances to be performed on other variants.

The final result stored in J_v contains a list of candidate tests for processing on each variant v. After completing calculation for all variants, we select variant v^* which has the largest number of tests executed on time. This variant v^* is then assigned to prototype i and tests in J_{v^*} are scheduled. The remaining tests will be considered for scheduling on the additional prototypes.

In Algorithm 4, we use the number of executed tests as a single criterion in order to select the best prototype variant. However, we realize that some tests should have higher priority to be scheduled before considering other tests. Due to the availability time for building a new prototype, we should try to allocate tests with early latest start times to firstly-built prototypes. Otherwise, after missing their due dates, the test can only be neglected.

Therefore, we propose Algorithm 5 in which the variant selection process also depends on the priority of tests. We include the step of searching for job j^*, the unscheduled job which has the minimum latest start time. Also, instead of considering all variants in V_r we limit ourselves to variants which are able to perform job j^*. To ensure that job j^* is executed without delay on the current prototype, it is the first test we add to set J_m.

Algorithm 4 Trade-off procedure between the number of executed tests and prototypes

```
Solve the set covering MILP model to determine $V_r$ from $V$
$J_a := J$
$J_n := \emptyset$
$i := 1$
repeat
  for all $j \in J_a$ do
    if $d_j - p_j < a_i + \min_{v \in V_r \cap M_j} \{s_v\}$ then
      $J_n := J_n \cup \{j\}$
      $J_a := J_a \setminus \{j\}$
    end if
  end for
  for all $v \in V_r$ do
    $J_v := \emptyset$
    $J_s := \{j \in J_a \cap N_v | \sum_{v \in V} M_j = 1\}$
    $J_m := \emptyset$
    Solve the CP-Late-Single
    for all $j \in J_s$ do
      if $C_j \leq d_j$ then
        $J_m := J_m \cup \{j\}$
      end if
    end for
    $J_s := J_a \cap N_v$
    Repeat solving the CP-Late-Single
    for all $j \in J_s$ do
      if $C_j \leq d_j$ then
        $J_v := J_v \cup \{j\}$
      end if
    end for
  end for
  Select variant $v^*$ which has the largest number of tests executed on time
  $J_a := J_a \setminus J_{v^*}$
  $i := i + 1$
until $J_a = \emptyset$
```

We implement the procedure using OPL Studio 3.7 [25]. As we need to frequently solve a great number of CP problems in each iteration, we do not wait until the solver reaches an optimal solution. We accept a feasible solution obtained within a specified time limit of 60 seconds.

The computational results for the instances with 41 and 100 tests are shown in Table 5.6, while Figures 5.4 and 5.5 present the solutions of the instances with 231 and 487 tests. For each prototype, we accumulate the number of executed tests and compare the solutions obtained

Algorithm 5 Trade-off procedure between the number of executed tests and prototypes with priority of tests

```
Solve the set covering MILP model to determine $V_r$ from $V$
$J_a := J$
$J_n := \emptyset$
$i := 1$
repeat
  for all $j \in J_a$ do
    if $d_j - p_j < a_i + \min_{v \in V_r \cap M_j}\{s_v\}$ then
      $J_n := J_n \cup \{j\}$
      $J_a := J_a \setminus \{j\}$
    end if
  end for
  Select test $j^*$ which has the minimum latest start time in $J_a$
  for all $v \in V_r \cap M_{j^*}$ do
    $J_v := \emptyset$
    $J_s := \{j \in J_a \cap N_v | \sum_{v \in V} M_j = 1\}$
    $J_m := \{j^*\}$
    Solve the CP-Late-Single
    for all $j \in J_s$ do
      if $C_j \leq d_j$ then
        $J_m := J_m \cup \{j\}$
      end if
    end for
    $J_s := \{j \in J_a \cap N_v\}$
    Repeat solving the CP-Late-Single
    for all $j \in J_s$ do
      if $C_j \leq d_j$ then
        $J_v := J_v \cup \{j\}$
      end if
    end for
  end for
  Select variant $v^*$ which has the largest number of tests executed on time
  $J_a := J_a \setminus J_{v^*}$
until $J_a = \emptyset$
```

from Algorithms 4 and 5 with the results known from Section 5.4 using the Slack/MinId algorithm.

We realize that the final outcome seems to be useless if our procedures need more prototypes than necessary to execute all tests. In the second instance, for example, the new solution obtained from Algorithm 5 requires 7 prototypes, although we previously knew that 5 prototypes are sufficient to carry out all 100 tests.

However, the results of instances with 231 and 487 tests show that our procedures can quickly execute more tests compared to the original strategy Slack/MinId. Only in the end, the trends become flat and steady as now we need to produce prototype variants which are quite specific for a few special tests.

As a result, the manufacturer has more chances of earlier detecting any failure which may occur in between. It is quite useful for the long-term planning where a large number of prototypes and tests is involved. Although applying these trade-off procedures may require a few prototypes more, we gain more flexibility to correct and repeat some tests.

Notice also that when solving the instance of 487 tests, Algorithm 4 behaves too greedy at the beginning, as few tests are finally neglected and cannot be completed on time. Algorithm 5 slightly compromises the number of executed tests, while being able to finish all tests within their due dates.

Table 5.6: Trade-off between the number of executed tests and prototypes for the instances with 41 and 100 tests.

n	number of prototypes							
	1	2	3	4	5	6	7	8
Slack/MinId								
41	17	19	37	39	41	-	-	-
100	39	61	78	98	100	-	-	-
Algorithm 4								
41	29	35	37	39	41	-	-	-
100	19	48	71	93	96	98	99	100
Algorithm 5								
41	29	34	37	39	41	-	-	-
100	18	36	63	90	97	99	100	-

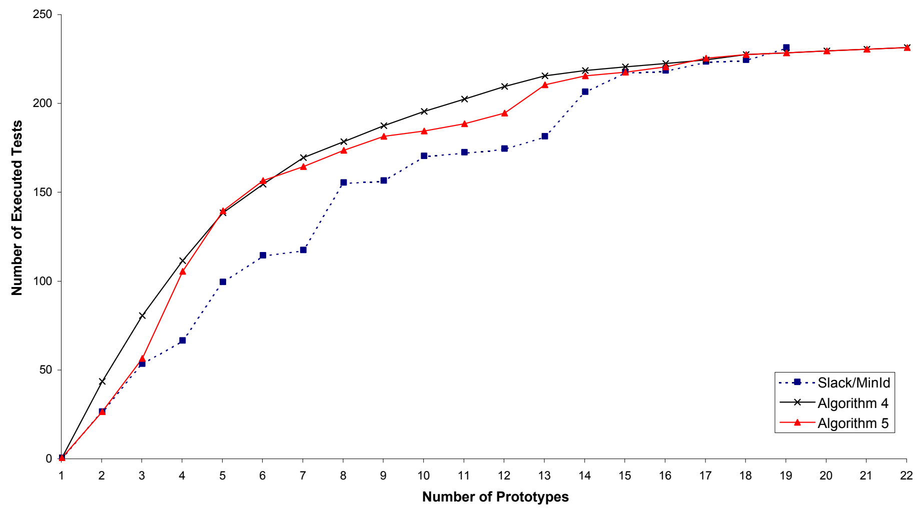

Figure 5.4: Trade-off between the number of executed tests and prototypes for the instance with 231 tests.

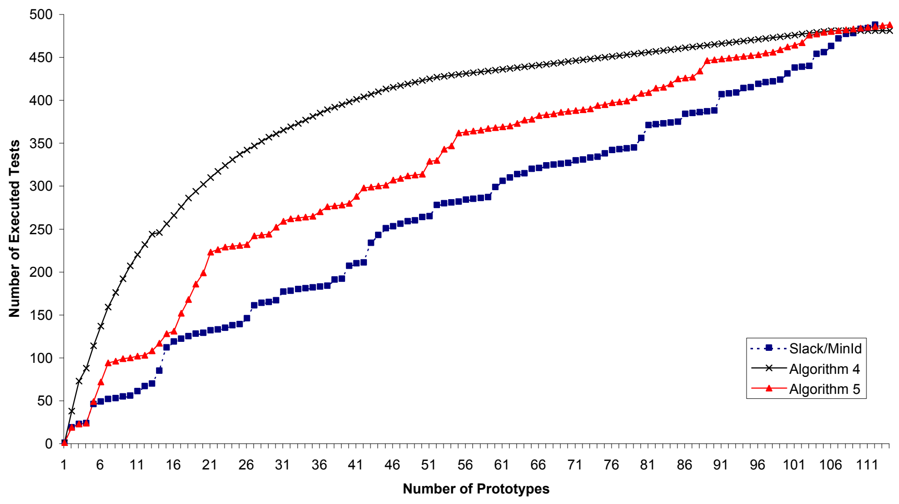

Figure 5.5: Trade-off between the number of executed tests and prototypes for the instance with 487 tests.

Chapter 6

Hybrid Approach Based on Bender's Decomposition

We start the description of our hybrid approach with the master MILP model to solve the planning aspect, before describing the slave CP model to find a complete schedule. Also, we suggest several Bender's cut constraints to eliminate invalid solutions from the previous steps.

Moreover, we suggest two alternative formulations of our hybrid approach. We allow CP to immediately correct some decisions determined by the MILP model, instead of waiting for the next iteration. As a result, the stronger Bender's cuts can be further developed. Finally, we compare various computational results obtained from solving the real-life instances by our hybrid methods.

6.1 Master MILP Model

To estimate the demand of prototypes in the master problem, we apply the lower bound formulation based on energetic reasoning suggested in Section 4.2.2. However, we must use binary variables to represent the number of prototypes instead of integer variables in order to comply with the logic-based Bender's cut.

Therefore, for each variant v, we replace its demand variable w_v with an array of binary variables $w_{v,h}$, where $h \in H = \{1, ..., g\}$. We assume at most g prototypes can be built for each variant. Also, at most one element of each array can be equal to 1. We define

$$w_{v,h} = \begin{cases} 1 & \text{if exactly } h \text{ prototypes of variant } v \text{ are required} \\ 0 & \text{otherwise} \end{cases}$$

For instance, there are 3 variants with the maximum limit of 5 prototypes. When the first and third variant require 1 and 4 prototypes, we obtain

$$w_{v,h} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

We keep using the test-to-variant allocation variables $y_{v,j}$ and the prototype variant variables $z_{v,i}$. After slightly modifying the energetic reasoning model, we obtain the master MILP model:

$$\text{Minimize} \quad \sum_{v \in V} \sum_{h \in H} h w_{v,h} \tag{6.1}$$

subject to

$$\sum_{h \in H} w_{v,h} \leq 1 \quad \forall v \in V \tag{6.2}$$

$$\sum_{v \in M_j} y_{v,j} = 1 \quad \forall j \in J \tag{6.3}$$

$$\sum_{h \in H} w_{v,h} \geq y_{v,j} \quad \forall j \in J, \forall v \in V \tag{6.4}$$

$$\sum_{v \in V} z_{v,i} \leq 1 \quad \forall i \in I \tag{6.5}$$

$$\sum_{i \in I} z_{v,i} = \sum_{h \in H} h w_{v,h} \quad \forall v \in V \tag{6.6}$$

$$\begin{aligned}
\sum_{h \in H} h w_{v,h}(t_2 - t_1) \geq & \sum_{j \in J(t_1,t_2)} y_{v,j} p_j + \sum_{\substack{j \in J_l(t_1,t_2) \cup \\ J_{ol}(t_1,t_2)}} y_{v,j}(r_j + p_j - t_1) + \cdots \\
& + \sum_{\substack{j \in J_r(t_1,t_2) \cup \\ J_{or}(t_1,t_2)}} y_{v,j}(t_2 - d_j + p_j) + \sum_{j \in J_{oa}(t_1,t_2)} y_{v,j}(t_2 - t_1) + \cdots \\
& + \sum_{j \in J_{\text{Last}} | d_j < t_1} y_{v,j}(t_2 - t_1) + \sum_{\substack{i \in I | t_1 < a_i + s_v < t_2, \\ a_i + s_v \geq b_v}} z_{v,i}(a_i + s_v - t_1) + \cdots \\
& + \sum_{\substack{i \in I | t_1 < b_v < t_2, \\ a_i + s_v < b_v}} z_{v,i}(b_v - t_1) + \sum_{\substack{i \in I | a_i + s_v \geq t_2, \\ a_i + s_v \geq b_v}} z_{v,i}(t_2 - t_1) + \cdots \\
& + \sum_{\substack{i \in I | b_v \geq t_2, \\ a_i + s_v < b_v}} z_{v,i}(t_2 - t_1) \quad \forall v \in V, \forall j \in N_v, \forall t_1 \in \bar{R}, \forall t_2 \in \bar{D}
\end{aligned} \tag{6.7}$$

$$t_2 - t_1 \geq \sum_{k \in J_s(j) \cap J(t_1,t_2)} y_{v,k} p_k \quad \forall v \in V, \forall j \in N_v, \forall t_1 \in \bar{R}, \forall t_2 \in \bar{D} \tag{6.8}$$

$$y_{v,j} = y_{v,k} \quad \forall j \sim k, \forall v \in V \tag{6.9}$$

$$\sum_{h \in H} h w_{vh} \geq y_{v,j} + y_{v,k} \quad \forall j \asymp k, \forall v \in V \tag{6.10}$$

$$\sum_{h \in H} h w_{v,h} \geq \sum_{j \in N_v \cap J_{\text{Last}}} y_{v,j} \quad \forall v \in V \tag{6.11}$$

In principle, the objective function (6.1) and (4.22) are the same. We can see also that Constraints (6.3)–(6.11) are equivalent to Constraints (4.23)–(4.31). Only Constraint (6.2) is included to ensure that at most one variable in the array H can equal one.

6.2 Slave CP Model

The solution of the master MILP model consists of $y_{v,j}$ and $z_{v,i}$ which determine the allocation of tests to variants and the sequence of prototype variants. The slave CP model is required only to determine the allocation of tests to prototypes x_j and the start time t_j.

Note that we must specifically determine the test allocation to the prototypes not just to the variants because of the constraints for tests executed on the same and different prototypes. Otherwise, we can basically consider a group of prototypes with the same variant as one cumulative resource with the capacity $C_v = \sum_{h \in H} hw_{v,h}$.

We further apply the CP model in Section 5.1 to formulate the slave CP model:

solve

$$y_{v_i,j} \neq 1 \Rightarrow x_j \neq i \quad \forall i \in I, \forall j \in J \tag{6.12}$$

$$z_{v,i} = 1 \Rightarrow v_i = v \quad \forall i \in I, \forall v \in V \tag{6.13}$$

$$t_j \geq r_j \quad \forall j \in J \tag{6.14}$$

$$t_j + p_j \leq d_j \quad \forall j \in J \tag{6.15}$$

$$\text{cumulative}(t_j | x_j = i, p_j | x_j = i, c_j = 1 | x_j = i, 1) \quad \forall i \in I \tag{6.16}$$

$$x_j = i \Rightarrow t_j \geq a_i + s_{v_i} \quad \forall i \in I,\ \forall j \in J \tag{6.17}$$

$$x_j = i \Rightarrow t_j \geq b_{v_i} \quad \forall i \in I,\ \forall j \in J \tag{6.18}$$

$$t_j + p_j \leq t_k \quad \forall j \prec k,\ j,k \in J \tag{6.19}$$

$$x_j = x_k \quad \forall j \sim k,\ j,k \in J \tag{6.20}$$

$$x_j \neq x_k \quad \forall j \asymp k,\ j,k \in J \tag{6.21}$$

$$x_j = x_k \Rightarrow \ t_j \geq t_k + p_k \quad \forall j \in J_{\text{Last}}, \forall k \in J, j \neq k\ . \tag{6.22}$$

As most constraints remain the same, we only explain the modifications here. First, we neglect the objective function (5.1) to minimize the peak demand since it is sufficient to find a feasible solution which satisfies all constraints.

Also, the results of variables $y_{v,j}$ from the master problem restrict tests to be performed just on the specified prototype variants, although there are other possible variants available. Therefore, we use Constraint (6.12) to replace the original component requirement which is

$$v_i \notin M_j \Rightarrow x_j \neq i \quad \forall i \in I,\ \forall j \in J. \tag{6.23}$$

Moreover, the sequence of prototype variants in the slave CP model must comply with the result from the MILP model as restricted by Constraint (6.13).

6.3 Bender's Cut

The Bender's cut constraint is proposed to eliminate invalid solutions from the previous steps. The constraint will be incorporated into the master MILP problem if the solutions from the previous steps are found infeasible by the slave CP model.

Assume that a solution of the MILP model at iteration k consists of $w^k_{v,h}$, $y^k_{v,j}$, and $z^k_{v,i}$. The simplest way to eliminate this solution is to include the *nogood* constraint

$$\sum_{v\in V}\sum_{h\in E_v^k} w_{v,h} + \sum_{v\in V}\sum_{j\in F_v^k} y_{v,j} + \sum_{v\in V}\sum_{i\in G_v^k} z_{v,i} \le \sum_{v\in V}(|E_v^k| + |F_v^k| + |G_v^k|) - 1 \tag{6.24}$$

where $E_v^k = \{h \in H | w_{v,h}^k = 1\}$, $F_v^k = \{j \in J | y_{v,j}^k = 1\}$, and $G_v^k = \{i \in I | z_{v,j}^k = 1\}$.

As it is quite inefficient to remove just one solution for each iteration, we try to concurrently eliminate other possible failures that might occur. To do so, we formulate sub-CP models for each prototype variant instead of solving just the slave CP model. Each sub-CP model considers only a cumulative resource constraint with its corresponding resource capacity and the set of allocated tests. Next, we check the feasibility of each sub-CP model separately. Assume the sub-CP model of variant v' is found infeasible. That means its resource capacity with the corresponding prototype sequence (specified in $E_{v'}^k$ and $G_{v'}^k$, respectively) is not sufficient to accommodate all the jobs in the given set, $j \in F_{v'}^k$. As the processing times are always constant, this result can be implied in any other variant. This leads to another Bender's cut

$$\sum_{h\in E_{v'}^k} w_{v,h} + \sum_{j\in F_{v'}^k} y_{v,j} + \sum_{i\in G_{v'}^k} z_{v,i} \le |E_{v'}^k| + |F_{v'}^k| + |G_{v'}^k| - 1 \quad \forall v \in V. \tag{6.25}$$

Notice that the complete schedule is invalid as well, if only one of all sub-CP models is not feasible. The *nogood* constraint (6.24) can be immediately included without solving the whole slave CP model. Although it takes an additional effort to solve the small sub-CP models, we expect to gain a benefit from reducing the number of iterations and the computation time for solving the large master MILP model.

Moreover, the prototype availability time can be considered as a dummy job competing with other normal jobs to occupy the resource. Assume the sub-CP model is infeasible when using one prototype in the second order of the prototype sequence. Due to the increasing availability time, it is also impossible to allocate the same set of tests to the prototype in the third order. Therefore, we can avoid a similar failure to happen on any succeeding prototype via the following constraint

$$\sum_{i\in I} a_i z_{v,i} < \sum_{i\in G_{v'}^k} a_i z_{v,i} + \bar{M}(|E_{v'}^k| + |F_{v'}^k| - \sum_{h\in E_{v'}^k} w_{v,h} + \sum_{j\in F_{v'}^k} y_{v,j}) \; \forall v \in V. \tag{6.26}$$

where $\bar{M}$ is a sufficiently large positive number, that is $\bar{M} = \sum_{i\in I} a_i$.

Constraint (6.26) is valid for resource capacity $E_{v'}^k$ and test allocation $F_{v'}^k$ and ensures that we have more time for executing these tests by using prototypes which are completed earlier.

Due to the restriction of the allocation of tests only to their specified variants, the hybrid approach in this section is called *the hybrid approach with fixed test allocation.* Its complete diagram is also provided in Figure 6.1.

6.4 Alternative Formulations of the Hybrid Approach

We further suggest two alternative formulations to apply the hybrid approach. First, assume we let the slave CP model choose any suitable variant for the tests. That means Con-

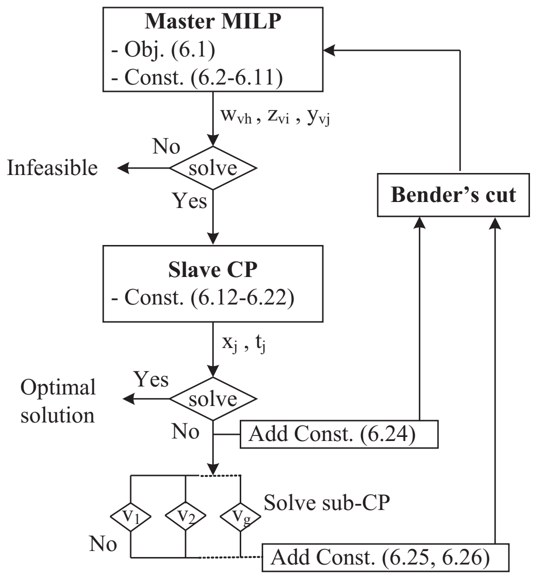

Figure 6.1: Diagram of the hybrid approach with fixed test allocation.

straint (6.12) is neglected. As a result, we can replace the *nogood* constraint (6.24) by a stronger Bender's cut

$$\sum_{v\in V}\sum_{h\in E_v^k} w_{v,h} + \sum_{v\in V}\sum_{i\in G_v^k} z_{v,i} \leq \sum_{v\in V}(|E_v^k| + |G_v^k|) - 1. \tag{6.27}$$

Due to Constraint (6.27), the master problem does not need to try many possibilities of allocation between tests and variants since we let CP take responsibility. Thus, the number of iterations can be reduced. However, this is reached at the cost of solving more difficult CP models.

In the second formulation, we allow CP to change the sequence of prototypes, while only the number of prototype variants is restricted by the master problem. To do so, the slave CP model must include the following constraint

$$\sum_{h\in H} hw_{v,h} = \sum_{i\in I|v_i=v} 1 \;\forall v \in V. \tag{6.28}$$

Constraint (6.28) ensures that the number of prototypes of variant v found in the slave CP equals its demand determined by the master MILP. After that, we introduce another Bender's cut constraint if the solutions are invalid.

$$\sum_{v\in V}\sum_{h\in E_v^k} w_{v,h} \leq \sum_{v\in V} |E_v^k| - 1. \tag{6.29}$$

Finally, we name both alternative formulations *the hybrid approach with fixed prototype sequence* and *the hybrid approach with fixed variant demand*, respectively, and present their diagrams in Figure 6.2.

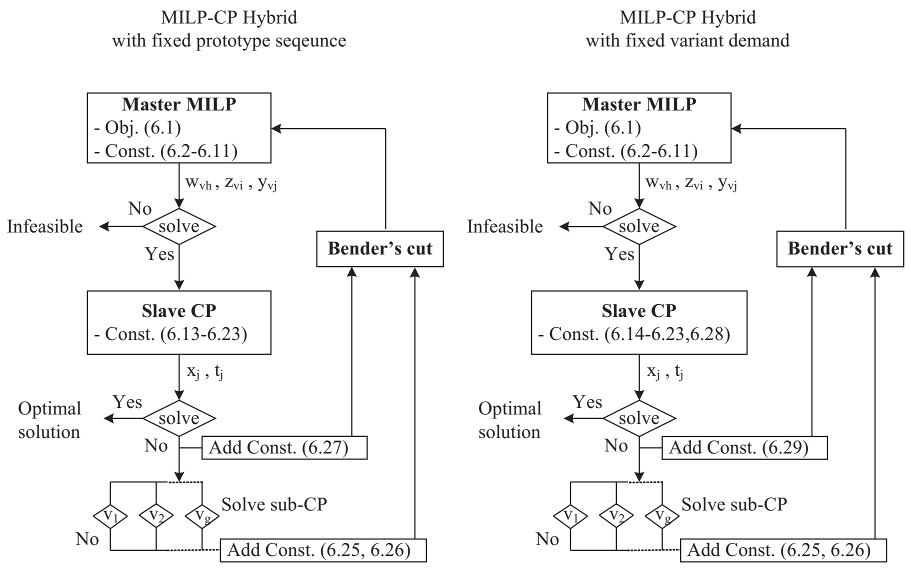

Figure 6.2: Diagrams of the hybrid approaches with fixed prototype sequence and fixed variant demand.

6.5 Computational Results

We apply the hybrid approach with time limits of 5 hours for total computation, 10 minutes for the master MILP and the slave CP, and 1 minute for the sub-CP. We solve the real-life instances with several scenarios. First, we assume not to consider the effect of the Left/Right-Shift term in the MILP problem. Next, we use the original hybrid approach with fixed test allocation, before applying the hybrid approaches with fixed prototype sequence and with fixed variant demand. We present the computational results in Table 6.1.

For all cases and all iterations, the master MILP can be solved optimally and the feasibility of the sub-CP problems can be proved within the time limits. However, sometimes a slave CP model can neither be proved feasible or infeasible. If this happens, we add a cut to eliminate this solution before proceeding to the next iteration. We clearly make a difference between the number of problems which are proved infeasible (#Inf) and which are unsolvable by our algorithms within the given time limits (#Uns).

We provide the solutions of the MILP model at the first and the last iteration (m_r^1, m_r^f). If the program can convert this solution to a complete solution within the overall time limit, m_r^f represents the optimal solution. Otherwise, m_r^f denotes the best lower bound as shown in the instances with 100 and 231 tests. The hybrid approach can find the optimal solution for our largest instance with 110 prototypes, compared with a solution with 111 prototypes obtained from the CP approach.

Also, we can solve the instance with 231 tests only after including the effect from the Left/Right Shift term. During the computation, there are a number of unsolvable slave CP models, which might lead to an optimal solution if more computation time was provided. Although we did not completely explore these intermediate solutions, we maintained optimality as the objective value m_r^f did not change since the first iteration.

Moreover, it took just 1 and 4 iterations to solve the instances with of 231 and 487 tests if we let the CP approach try to relocate tests to any suitable variant instead of using only the variant specified by the master problem. Note that Constraint (6.12) restricts the slave CP model not to change the misleading allocations which must be corrected later after repeating several iterations. The CP approach is more efficient to search for other feasible alternatives while MILP is applied only to estimate the demand and determine the sequence of prototype production.

Also, we can get a complete solution requiring 5 prototypes for the instance with 100 tests. However, we cannot be sure that this is an optimal solution since for the objective of 4 prototypes, the slave CP model cannot be completely solved within the time limit in several iterations.

However, when we let the CP approach change a prototype sequence, the instance with 487 tests cannot be solved as before. When we use less information from the planning master, it becomes harder for the slave CP model to find a feasible solution with 110 prototypes. Remember that a solution with one additional prototype is found if we solve the complete problem by using CP alone.

Therefore, the hybrid approach can optimally solve a very large problem but fails to solve a smaller problem. We assume that the characteristics of data play a more important role than the problem size. Even for a large problem, the overall problem can be clearly divided via our decomposition method when tests are naturally separated to their specific variants and along the time horizon by the component requirements and the temporal constraints. For the instance with 100 tests, we found that almost all tests can be assigned to any prototype variant. This leads to a huge number of possible reallocations which cannot be easily eliminated by using our Bender's cuts.

As a stronger cut policy can be applied if we know a minimum set of jobs which causes the conflict, we tried the algorithm suggested by Junker [30]. Given a set of jobs, this algorithm can determine the critical set within a polynomial number of steps by solving partitioned problems. Although these partitioned problems are significantly smaller, many of them still cannot be solved within reasonable time. Only a few cuts can be generated and they are not enough to help us to obtain an optimal solution.

Table 6.1: Minimizing the number of required prototypes using the hybrid approach.

n	Master MILP				Slave CP			Sub CP	
	# It	m_r^1	m_r^f	Time(s)	#Inf	#Uns	Time(s)	#Inf	Time(s)
Neglecting Left/Right-Shift in master MILP									
41	1	5	5[a]	0.77	0	0	0.01	0	0.01
100	183	4	4[b]	73.08	154	29	18,076.10	147	0.08
231	56	18	18[b]	18.6	26	30	18,023.60	27	0.07
487	19	110	110[a]	1,000.4	18	0	0.02	28	0.26
Hybrid approach with fixed test allocation									
100	191	4	4[b]	75.90	161	30	18,344.4	128	0.14
231	13	18	18[a]	3.86	0	12	7,213.4	0	0.01
487	4	110	110[a]	229.38	3	0	0.01	5	0.03
Hybrid approach with fixed prototype sequence									
100	6	4	5	1.55	2	3	1,801.49	2	0.01
231	1	18	18[a]	0.29	0	0	0.50	0	0.01
487	4	110	110[a]	230.15	3	0	0.61	5	0.07
Hybrid approach with fixed variant demand									
100	2	4	5	0.41	0	1	600.41	0	0.01
231	1	18	18[a]	0.26	0	0	0.45	0	0.01
487	40	110	110[b]	2,079.46	15	25	15,012.5	22	0.64

[a] Optimal Solution
[b] Lower Bound
m_r^1, m_r^f: solution of MILP at the first and last iteration
#It: number of total iterations
#Inf: number of infeasible problems
#Uns: number of unsolvable problems

Chapter 7

Performance and Robustness Verification

In the previous chapters we applied our approaches to solve a small number of real-life instances. In order to investigate the performance and robustness of our algorithms, we use random instances generated for a similar problem. As the new problem has some different requirements, we first explain the changes necessary for our formulations.

Moreover, we briefly discuss a concept of the heuristic method which was applied to solve the instances. Also, we suggest more search schemes based on the combinations of different resource and job selection methods before ending with the comparison of results obtained from various approaches.

7.1 The New Problem

As mentioned before, Bartels and Zimmermann [6] studied a test scheduling problem which is similar to our case in several aspects. First, it considers the same objective function to schedule all the tests using the minimum number of prototypes. Also, due to the component requirements, the suitable variant of prototypes must be chosen from the given list to carry out the tests.

The temporal constraints are more generally regarded as the minimum and maximum time lags between the start times of activities. As a result, we include the condition $t_j - t_k \geq \delta_{kj}$, where δ_{kj} is a weight parameter to represent the necessary time lag between the start times of both activities. The overall project deadline $\bar{d}$ is given. Also, we include two dummy jobs 0 and $n+1$ which have zero processing times and must be executed at time 0 and at the deadline. The release and due dates of job i can be represented by the minimum time lag between jobs 0 and i, and between jobs i and $n+1$, respectively.

Moreover, the precedence constraint can be realized as the time lag constraint by setting the weight parameter to a processing time of the predecessor job. That means we have $t_k - t_j \geq p_j$ for $j, k \in J : j \prec k$. The prototype availability time a_i is still considered because of the limited capacity to build new prototypes, while the set-up time s_{v_i} and the component delay time b_{v_i} are neglected.

The constraints for the crash tests are still the same, while there is no need for tests to be executed on the same or on different prototypes. As mentioned before, a new restriction for a partially ordered destructive test is required. We state that for two different tests

$j, k \in J$: $j \simeq k$, the constraint is valid iff either job k precedes job j or both jobs are executed on different prototypes. That means the execution of test j can cause partial damage such that an associated prototype is no longer valid for test k. However, we can use this prototype to perform other tests which require its undamaged components.

7.1.1 Formulation for CP Approach

In the following, we provide the CP model for the new problem.

$$\text{Minimize} \quad \max_{j \in J} x_j$$

subject to

$$t_j - t_k \geq \delta_{kj} \quad \forall j, k \in J \cup \{0, n+1\}, j \neq k \tag{7.1}$$

$$x_j = x_k \Rightarrow t_j \geq t_k + p_k \quad \forall j \simeq k, j, k \in J \tag{7.2}$$

$$x_j = i \Rightarrow t_j \geq a_i \quad \forall i \in I, \forall j \in J \tag{7.3}$$

$$v_i \notin M_j \Rightarrow x_j \neq i \quad \forall i \in I, \forall j \in J \tag{7.4}$$

$$\text{cumulative}(t_j | x_j = i, p_j | x_j = i, c_j = 1 | x_j = i, 1) \quad \forall i \in I \tag{7.5}$$

$$x_j = x_k \Rightarrow \; t_j \geq t_k + p_k \quad \forall j \in J_{\text{Last}}, \forall k \in J, j \neq k \tag{7.6}$$

In comparison with the CP model in Section 5.1, we here include the new requirement for the time lag constraint as shown in Constraints (7.1). Also, Constraint (7.2) becomes necessary when there is a partially ordered destructive restriction for two different tests $j, k \in J$: $j \simeq k$. When both tests happen to share the same prototype, we have to ensure that test k is executed before test j.

Constraint (7.3) for the restriction on the prototype availability time is just slightly changed from Contraint (5.6) by neglecting the variant set-up times. The rest of the new model remains the same. Constraint (7.4) is introduced for the component requirement. The resource constraint is enforced by Constraint (7.5). For the crash test, we need Constraint (7.6).

7.1.2 Formulation for Hybrid Approach

To apply the hybrid approach, we have to determine both the master MILP and the slave CP models.

For the master problem, we can keep using the model in Section 6.1. Only the additional Constraints (6.9) and (6.10) for tests executed on the same and different prototypes can be neglected.

Unfortunately, we cannot even partially consider the impact of both time lag constraint and the partially ordered test. It is hard to include this kind of precedence relation because our model is based on energetic reasoning, where we need to know the exact release and due dates.

For the slave CP model, we can apply most of the constraints formulated in the previous section. Thus, we will explain only the different points. Since there exist three different ways of using the hybrid approach, we further describe each case as follows.

1. Slave CP model for the hybrid approach with fixed test allocation

solve

$$y_{v_i,j} \neq 1 \Rightarrow x_j \neq i \quad \forall i \in I, \forall j \in J \tag{7.7}$$
$$z_{v,i} = 1 \Rightarrow v_i = v \quad \forall i \in I, \forall v \in V \tag{7.8}$$
$$t_j - t_k \geq \delta_{kj} \quad \forall j,k \in J \cup \{0, n+1\}, j \neq k \tag{7.9}$$
$$x_j = x_k \Rightarrow t_j \geq t_k + p_k \quad \forall j \simeq k, j,k \in J \tag{7.10}$$
$$x_j = i \Rightarrow t_j \geq a_i \quad \forall i \in I, \forall j \in J \tag{7.11}$$
$$\text{cumulative}(t_j|x_j = i, p_j|x_j = i, c_j = 1|x_j = i, 1) \quad \forall i \in I \tag{7.12}$$
$$x_j = x_k \Rightarrow \ t_j \geq t_k + p_k \quad \forall j \in J_{\text{Last}}, \forall k \in J, j \neq k \tag{7.13}$$

Constraint (7.7) restricts the allocation of tests to prototypes which belong to the variants specified by the $y_{v_i,j}$ variables in the master model. Similarly, Constraint (7.8) ensures that the results of $z_{v,i}$ variables determine the variant of each prototype in the slave CP model.

2. Slave CP model for the hybrid approach with fixed prototype sequence

$$\text{solve} \tag{7.14}$$
$$z_{v,i} = 1 \Rightarrow v_i = v \quad \forall i \in I, \forall v \in V \tag{7.15}$$
$$t_j - t_k \geq \delta_{kj} \quad \forall j,k \in J \cup \{0, n+1\}, j \neq k \tag{7.16}$$
$$x_j = x_k \Rightarrow t_j \geq t_k + p_k \quad \forall j \simeq k, j,k \in J \tag{7.17}$$
$$x_j = i \Rightarrow t_j \geq a_i \quad \forall i \in I, \forall j \in J \tag{7.18}$$
$$v_i \notin M_j \Rightarrow x_j \neq i \quad \forall i \in I, \forall j \in J \tag{7.19}$$
$$\text{cumulative}(t_j|x_j = i, p_j|x_j = i, c_j = 1|x_j = i, 1) \quad \forall i \in I \tag{7.20}$$
$$x_j = x_k \Rightarrow \ t_j \geq t_k + p_k \quad \forall j \in J_{\text{Last}}, \forall k \in J, j \neq k \tag{7.21}$$

To allow CP to select any appropriate prototype variant, we use Constraint (7.4) for the component requirement, while Constraint (7.7) is removed.

3. Slave CP model for the hybrid approach with fixed variant demand

$$\text{solve} \tag{7.22}$$
$$\sum_{h \in H} hw_{v,h} = \sum_{i \in I | v_i = v} 1 \quad \forall v \in V \tag{7.23}$$
$$t_j - t_k \geq \delta_{kj} \quad \forall j,k \in J \cup \{0, n+1\}, j \neq k \tag{7.24}$$
$$x_j = x_k \Rightarrow t_j \geq t_k + p_k \quad \forall j \simeq k, j,k \in J \tag{7.25}$$
$$x_j = i \Rightarrow t_j \geq a_i \quad \forall i \in I, \forall j \in J \tag{7.26}$$
$$v_i \notin M_j \Rightarrow x_j \neq i \quad \forall i \in I, \forall j \in J \tag{7.27}$$
$$\text{cumulative}(t_j|x_j = i, p_j|x_j = i, c_j = 1|x_j = i, 1) \quad \forall i \in I \tag{7.28}$$
$$x_j = x_k \Rightarrow \ t_j \geq t_k + p_k \quad \forall j \in J_{\text{Last}}, \forall k \in J, j \neq k \tag{7.29}$$

Constraint (7.23) ensures that the results obtained from the master problem determine the number of prototypes for each variant in the slave model. However, the production sequence of these prototypes can be changed as Constraint (7.8) is removed.

Let us assume that there are 2 variants requiring 2 and 1 prototypes, respectively. CP will search for the production sequence which can lead to a feasible schedule. For instance, CP may try using variables $v_1 = 2$, $v_2 = 1$, and $v_3 = 1$. That means we build a prototype for the second variant first and then two prototypes of the first variant. Also, it may be possible for the production sequence with variables $v_1 = 1$, $v_2 = 2$, and $v_3 = 1$. We can see that CP commits only to keep the total demand of each variant unchanged.

7.1.3 The Existing Approach

After describing the models for the new problem, we provide more details about the approach previously used by Bartels and Zimmermann [6]. They applied the priority-rule heuristic method which tries to schedule the most critical job first by using criteria like the minimum latest start time or the highest total number of successors. Also, various kinds of planning schemes are applied to allocate the selected test, for instance, as early (or late) as possible in the forward (or backward) planning schemes. Each variation of different priority-rules and planning schemes is considered as a single-pass heuristic method. It can either find one or no feasible solution.

To further improve the method, the concept of multi-pass heuristic procedure is applied. Instead of relying on a limited number of rules, the priority can be calculated from the weighted sum of several rules. By repeatedly solving each instance with different weight values, we hope to achieve better feasible schedules. Also, we can apply various planning schemes. The whole technique is called the multi-directional multi-pass heuristic procedure.

Within this process, tests are assigned to the existing prototypes if possible. When a new prototype must be inevitably included, the algorithm tries to select the variant which is suitable for most of the unscheduled tests.

7.2 Additional Branch Schemes

As the performance of CP depends significantly on the search part, we introduce more branch schemes besides Slack/MinId and LST/MinId suggested in Section 5.2. In general, the search tree is so large that CP cannot visit all nodes within a given time limit. Using various branch schemes can help us explore other potential areas to find better solutions.

It is quite important to have an efficient strategy especially when CP starts exploring the search tree. Since we use the standard backtrack procedure, only nodes at the bottom of the tree are thoroughly visited. Therefore, we focus on changing our branch scheme only in the allocation part, while the other parts for assigning prototype variants and start times remain the same. In the following, we suggest several criteria to prioritize the tests and prototypes.

7.2.1 Test Selection Method

To select a critical test, we always consider the number of remaining possible prototypes as the first criterion. To break a tie, we apply several temporal criteria:

- Slack
  Tests are selected using the minimum slack time.

- LST
  Tests are selected using the minimum latest start time.

- Slack-LST
  Normally tests are selected using the minimum slack time. But when the slack times of some tests become less than a certain point compared with the prototype construction period a_p, we switch to consider those tests and select one of them using the minimum latest start time. The Slack-LST rule is formally described in Algorithm 6. By setting the parameter $c' = 1$, 2, and 3, we obtain the rules: Slack-LST1, Slack-LST2, and Slack-LST3, respectively.

  These rules allow us to occasionally schedule tests whose latest start times are getting tight since we might miss their due dates if only considering the slack time. Notice that the slack time represents the difference between the possible latest and earliest start times to execute a test. When the gap is quite small compared to the prototype availability time, it also means only few prototypes can perform the test. We assume that the resource availability becomes more critical for the test with minimum latest start time since it certainly needs prototypes built quite at the beginning of the production sequence. These prototypes tend to be overwhelmed because our policy suggest using them repeatedly to reduce the peak demand.

Algorithm 6 Test selection method: Slack-LST

```
%Let $J_a$ be the set of tests not yet allocated
Initialize $J_m := \emptyset$ and $J_t := \emptyset$
From $J_a$, find test $j$ having the minimum number of possible prototypes, and add to $J_m$
From $J_m$, find test $j$ whose $lst_j - est_j < c'a_p$, and add to $J_t$
if $J_t \neq \emptyset$ then
  From $J_t$, select test $j$ with the minimum latest start time
else
  From $J_m$, select test $j$ with the minimum slack time
end if
Return test $j$
```

- LST-Slack
  Normally tests are selected using the minimum latest start time. But when the slack times of some tests become less than a certain point compared with their processing times, we switch to consider those tests and select one of them using the minimum slack time. The LST-Slack rule is formally described in Algorithm 7. By specifying the parameter $\bar{c} = 1$, 2, and 3 in the algorithm, we obtain the rules: LST-Slack1, LST-Slack2, and LST-Slack3, respectively.

  These rules allow us to occasionally consider tests whose slack times are getting smaller compared to their processing times. Unlike the slack time, the latest start time neglects to consider the overall possible time gap to schedule tests. Since the rule normally selects a test with an early due date, we assume even the test with the minimum due date may not be so critical. Thus, we should occasionally switch to use the slack time as the measurement criterion.

Algorithm 7 Test selection method: LST-Slack

```
%Let $J_a$ be the set of tests not yet allocated
Initialize $J_m := \emptyset$ and $J_t := \emptyset$
From $J_a$, find test $j$ having the minimum number of possible prototypes, and add to $J_m$
From $J_m$, find test $j$ whose $lst_j - est_j < c'p_j$, and add to $J_t$
if $J_t \neq \emptyset$ then
  From $J_t$, select test $j$ with the minimum slack time
else
  From $J_m$, select test $j$ with the minimum latest start time
end if
Return test $j$
```

- Backward Planning
  We further implement a search scheme using problem-specific knowledge. As suggested by Bartels and Zimmermann [6], it is beneficial to apply a backward planning scheme especially in this case which has quite a lot of crash tests and partially destructive tests. The backward planning rule is formally described in Algorithm 8.

  Since we want to perform the tests as late as possible, the maximum latest start time is considered as the second criterion after the number of possible machines. Also, we prefer crash tests and partially ordered destructive tests to other tests.

Algorithm 8 Test selection method: Backward Planning

```
%Let $J_a$ be the set of tests not yet allocated
Initialize $J_m := \emptyset$ and $J_t := \emptyset$
From $J_a$, find test $j$ having the minimum number of possible prototypes, and add to $J_m$
From $J_m$, find test $j$ with the maximum latest start time, and add to $J_t$
if $J_t \cap J_{\text{Last}} \neq \emptyset$ then
  arbitrarily select a crash test $j$ in $J_t$
else if $J_t \cap J_d \neq \emptyset$, where $J_d = \{j | j, k \in J, j \simeq k\}$ then
  arbitrarily select a partially ordered destructive test $j$ in $J_t$
else
  arbitrarily select a test $j$ in $J_t$
end if
Return test $j$
```

7.2.2 Prototype Selection Method

Once a critical test is found, it must be allocated to an appropriate prototype. We apply two prototype selection methods trying to reuse the resources when possible in order to minimize the peak demand.

- MinId
  We select a prototype which can perform test j and has the minimum index i.

- MaxSt
  We prefer to select a possible prototype already required by any other test since using a

new prototype should be avoided. After that, we choose a prototype which can perform test j at the maximum latest start time. Unlike the MinId method, the MaxSt no longer tries to use the prototypes available early. These prototypes should be used only if necessary as the number of prototypes is especially scarce at the beginning.

The MaxSt rule is formally described in Algorithm 9. As mentioned before, during the procedure we select one test and allocate it to an appropriate prototype until all tests can be scheduled. We use set J_b to collect tests which are already allocated. These tests have their corresponding prototypes which are gathered in set I_r. These prototypes must be built as we have assigned the jobs to them. We prefer to use these prototypes again if one of them can execute test j. That is the case when $I_p \cap I_r \neq \emptyset$, where I_p is a set of all prototypes which can perform test j. We then select a prototype by considering the maximum latest start time.

We have to mention further that I_p contains also every prototype in $I \cap I_r$. Until now, these prototypes are not yet required to perform the tests in J_b. However, when $I_p \cap I_r = \emptyset$, we must inevitably allocate test j to one of the unused prototypes. The minimum index policy is applied for the prototype selection.

Algorithm 9 Prototype selection method: MaxSt

```
%Given the selected test $j$ and the set of allocated tests $J_b$
%Let $I_p$ be the set of prototypes possible to execute test $j$
%Let $I_r$ be the set of prototypes required by test $j \in J_b$
if $I_p \cap I_r \neq \emptyset$ then
  Select prototype $i$ which can perform test $j$ at the maximum start time
else
  Select a new prototype $i \in I_p$ with the minimum index
end if
Return machine $i$
```

7.2.3 Resulting Branch Schemes for Test Allocation

We further combine the proposed test and prototype selection methods in order to achieve various allocation branch schemes. For instance, Algorithm 10 represents the Slack/MaxSt scheme. Function SelectTest(Slack, J_a) selects test j from a set of tests not yet allocated by using the minimum slack time. After that, function SelectPrototype(MaxSt, j, $J \cap J_a$) finds prototype i to perform test j by applying the MaxSt selection method.

Algorithm 10 Slack/MaxSt test allocation branch scheme

```
%PART 1) Assign Tests to Prototypes
%Let $J_a$ be the set of tests not yet allocated
$J_a := J$
while $J_a \neq \emptyset$ do
  $j$ = SelectTest(Slack, $J_a$)
  $i$ = SelectPrototype(MaxSt, $j$, $J \cap J_a$)
  Try
    $x_j = i$
  or
    $x_j \neq i$ (for backtracking)
  $J_a = J \setminus \{j\}$
end while
```

7.3 Computational Results

For this new problem, Bartels and Zimmermann [6] generated random instances based on the real characteristics, for instance, 10% and 30% are the crash tests and partially ordered destructive tests which have an impact on 50% of all tests. There are two sets of instances: 100 instances of 20 tests with 4 variants; and 60 instances of 600 tests with 25 variants. All instances are available on their web-site (http://www.wiwi.tu-clausthal.de/testsets-evt).

7.3.1 Solving the Random Instances of 20 Tests

Bartels and Zimmermann [6] reported that the small instances can be solved optimally using the MILP method within 100 seconds on Pentium IV, 2.4 GHz, while the best solution of their heuristic approach is about 8% far from the optimal values.

To evaluate the performance of the CP approach, we first solve the small instances and can achieve the same optimal solutions for all cases as shown in Table A.5. The average computation time is 8.52 and 10.25 seconds for our search algorithms Slack/MindId and LST/MinId, respectively. In fact, we cannot compare the performance with MILP when using different computer platforms. However, we may conclude that both methods are still quite efficient when dealing with the small problems.

We further apply the hybrid approach and its alternative formulations. However, only the hybrid approach with fixed variant demand can obtain the optimal solutions with an average computation time of 11.22 seconds and about 18 iterations. We realize that our MILP model provides many misleading directions as it fails to consider an impact of several constraints in the scheduling problem. Even for these small problems, it takes a number of iterations to correct the mistakes. These mistakes cannot even be recovered when the slave CP models excessively obey the planning part. In this situation, it is better to let CP take more responsibility in order to complement the MILP model.

7.3.2 Solving the Random Instances of 600 Tests

For the large instances, Bartels and Zimmermann [6] can apply only the priority-rule heuristic method. It takes less than 0.4 seconds for the single-pass method and totally around 30 seconds for the multi-directional multi-start method which actually applies the single-pass heuristic procedure to solve each instance 100 times with different weight values.

We further apply CP with various search schemes to solve the large instances of 600 tests. We first consider the test selection methods: LST, Slack, Slack-LST, and LST-Slack, together with MinId and MaxSt rules for the prototype selection. The total number of prototypes is initialized at $m = \lceil n/3 \rceil$.

After getting all solutions, we can determine the best feasible solutions without backward planning. Next, we subtract these results by one prototype to initialize the number of machines m for Backward Planning/MinId and Backward Planning/MaxSt.

The detailed computation results are shown in Table A.6-A.10. The main results are summarized in Table 7.1 where the solutions obtained from CP are compared with the heuristic method.

In addition, Table 7.2 further provides the performance analysis of using different search schemes. Of totally 60 instances, we present the number of instances which can be solved and the number of instances whose solutions are improved compared with those obtained by the heuristic method. Also, we provide a number of instances whose solutions are equal to the best feasible results known from all methods. The average value of the corresponding deviation gaps can be further determined.

First when comparing our basic search algorithms: Slack/MinId and LST/MinId, we can achieve feasible solutions for 56 and 55 instances with deviation gaps of around 1.89% and 5.94%, respectively. Using the slack time can solve slightly more instances with a smaller deviation gap, while contributing the best solutions for 17 cases. Although neither algorithm Slack/MinId nor algorithm LST/MinId is perfectly robust to cope with all cases, both strategies can complement each other and achieve better solutions for 38 instances compared with the heuristic method.

Moreover, we consider the results from the combined strategies: LST-Slack and Slack-LST with the setting parameter $c' = 1, 2, 3$. Although we try to simultaneously apply both slack and latest start times, we cannot successfully combine LST/MinId and Slack/MinId to achieve a single algorithm which can handle all instances. For the best case, using Slack-LST2/MaxSt still misses one instance.

We do not further try to set parameter c' with different values since we noticed that the number of instances solved does not keep increasing after changing from Slack-LST2 and LST-Slack2 to Slack-LST3 and LST-Slack3, respectively.

Regarding the computation time, CP takes around five minutes on average to achieve the best solutions. For the remaining computation time of around 1 hour, CP got lost in the huge search tree. We also notice that the backtracking procedure always happens as our branch scheme does not directly guide to the solutions. However, we should not wait too long and only rely only on backtracking. It is better to stop and restart computation with the different search schemes allowing us to explore other possibilities.

Before applying the backward planning scheme, CP has already improved the solutions for 46 instances when compared with the heuristic method. We then use the best solutions from our CP to initialize the number of prototypes available for the backward planning scheme. As a result, we can find better solutions for 11 instances more. Mostly, we further improve our own best results, while the number of instances whose solutions are improved compared to those obtain from the heuristic method slightly increase to 48.

After that, we calculate the average number of required prototypes for these 60 instances using all solving methods available. The priority-rule heuristic method finds the solutions with an average of 74.83 prototypes. We can reduce the number to 73.37 prototypes using Slack-/MinId and LST/MinId. The number further decreases to 72.12 prototypes after applying other search schemes.

Moreover, we determine the difference between the number of required prototypes obtained from CP and from the heuristic method, that is, $m_r^{\text{CP}} - m_r^{\text{Heu}}$. The histogram is shown in Figure 7.1. Similarly, we further compare CP with the best known results, as shown in Figure 7.2.

In most cases, CP can decrease the prototype demand. Nevertheless, there are 2 instances where the heuristic method finds better solutions than CP. We may find out which heuristic rule can lead to these solutions and implement it as another search scheme in CP.

Our hybrid approach fails to achieve complete solutions for all large instances. The average lower bound is just 60.42 or differs by around 20.5% from the best feasible solutions. Mostly, the lower bounds are 60 due to the number of crash tests which is always 10% of the total number of tests. Only for the instance numbers 38, 40, and 56, we obtain lower bounds at 66, 67, and 72, respectively.

Obviously, our master problem cannot consider the impact of partially ordered destructive tests covering 30% of the total number of tests. Moreover, our model neglects the time lag constraints required for most of the tests.

We want to further mention another possible drawback of the hybrid approach. Notice that the size of our MILP model depends on the number of distinct values of both release and due dates. If each job always has its own values, the model can become too large to be solved. This situation may occur more often in the case of randomly generated instances, while in practice these values tend to be assigned in a duplicate fashion.

It is also possible to group all random values into fewer milestones in order to reduce the size of the formulated MILP problem. However, the performance of the hybrid approach can be diminished since the master model should be as precise as possible. Therefore, we might have to find a trade-off between the correctness of the master problem and the system capacity.

Table 7.1: Computational results from solving 60 instances of 600 tests using various approaches

No.	Heu	CP			No.	Heu	CP		
		LST-Slack	w/o Back	Best			LST-Slack	w/o Back	Best
1	63	62[a]	62	62	31	71	72	68[a]	68
2	94	89[a]	89	89	32	72	71	70[a]	70
3	88	84	83[a]	83	33	67	64	63[a]	63
4	71	70	69[a]	69	34	75	73[a]	73	73
5	68[a]	68[a]	68	68	35	72	68[a]	68	68
6	84	79	77[a]	77	36	83	79[a]	79	79
7	63[a]	67	63[a]	63	37	80	77	76	74[a]
8	97	98	98	96[a]	38	99	96	94[a]	94
9	66	63[a]	63	63	39	73	70[a]	70	70
10	70	69	69	66[a]	40	106	100	98[a]	98
11	63	63	63	62[a]	41	71	68[a]	68	68
12	69	69	68	66[a]	42	67[a]	68	67[a]	67
13	67	66	65[a]	65	43	67	62[a]	62	62
14	82	79	79	78[a]	44	64	64	62[a]	62
15	69	66	64[a]	64	45	84	82	80[a]	80
16	70	72	69[a]	69	46	69	66	65[a]	65
17	65	67	63[a]	63	47	76[a]	78	76[a]	76
18	86	82	81[a]	81	48	67	64[a]	64	64
19	72	69	68[a]	68	49	85	81	80[a]	80
20	62[a]	62[a]	62	62	50	76	75	73[a]	73
21	79	78	77[a]	77	51	79	76	76	74[a]
22	65[a]	67	66	66	52	98	93	93	92[a]
23	63[a]	63[a]	63	63	53	67	66	65	64[a]
24	65	63	63	61[a]	54	64	64	63[a]	63
25	66	66	65[a]	65	55	66[a]	66[a]	66	66
26	81[a]	83	82	81[a]	56	106	100	98[a]	98
27	69	65[a]	65	65	57	72	69[a]	69	69
28	65	66	64[a]	64	58	64	63	62[a]	62
29	87[a]	87[a]	87	87	59	95	92[a]	92	92
30	81[a]	83	81[a]	81	60	65[a]	70	69	69
					Avg.	74.83	73.37	72.42	72.12

[a] Best feasible solutions of all methods
Heu: Heuristics
LST-Slack: LST/MinId and Slack/MinId
w/o Back: Before apply backward planning
Best: Best solutions of CP

Table 7.2: Analysis of solving 60 instances of 600 tests using CP with various branch schemes

Job-selection method	Machine-selection method	Time(s)	#Solved instances	#Imp Heu	#Best	Avg. Gap from Best Solution (%)
Slack	MinId	268.63	56	37	17	1.89
	MaxSt	278.08	58	39	17	1.80
LST	MinId	273.83	55	13	1	5.94
	MaxSt	263.76	55	12	1	6.07
Slack-LST1	MinId	262.70	57	35	13	2.43
	MaxSt	257.01	56	36	10	2.51
Slack-LST2	MinId	264.11	58	37	18	1.93
	MaxSt	264.68	59	38	16	1.93
Slack-LST3	MinId	264.39	58	35	18	1.97
	MaxSt	268.08	58	36	18	1.82
LST-Slack1	MinId	267.44	58	27	6	3.94
	MaxSt	265.30	58	25	6	4.15
LST-Slack2	MinId	282.48	57	27	9	3.64
	MaxSt	260.85	58	27	8	3.83
LST-Slack3	MinId	277.48	56	34	15	2.24
	MaxSt	272.47	55	32	14	2.56
Backward Planning	MinId	66.35	18	48	57	0.15
	MaxSt	69.84	16	46	56	0.19
Scheduler		2,592.51	44	2	0	40.10
Best results by Slack+LST/MinId			60	38	18	1.89
Best results w/o Backward Planning			60	46	47	0.55
Best results from CP			60	48	58	0.13

#Imp Heu: number of instances whose solutions are better than those obtained by the heuristic method
#Best: number of instances whose solutions are equal to the best-known results

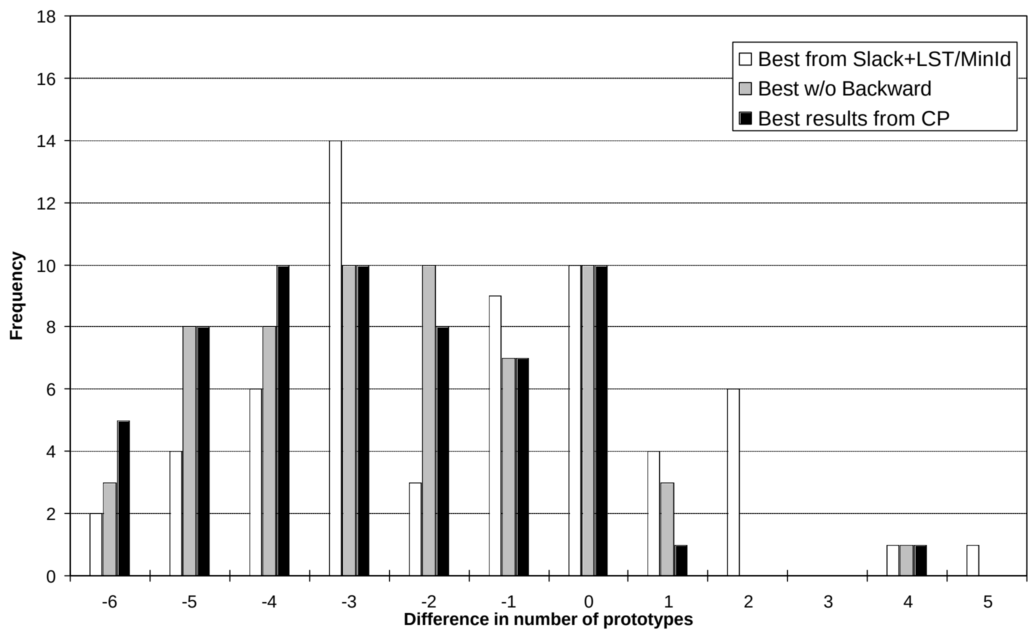

Figure 7.1: Histogram of the difference between the number of required prototypes obtained from CP and from the priority-rule heuristic method

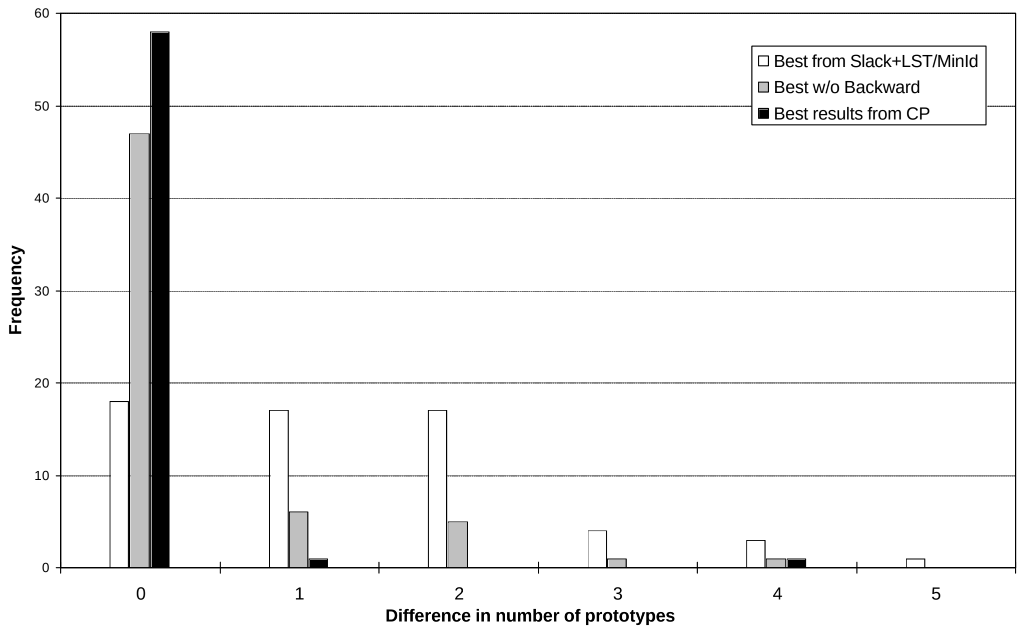

Figure 7.2: Histogram of the difference between the number of required prototypes obtained from CP and from the best known results

Chapter 8

Conclusion and Future Research

We finally summarize the main contributions of our work, before closing the thesis with open issues for future research.

8.1 Thesis Conclusion

We have presented a scheduling problem of the automotive industry where several hundreds of tests must be performed on vehicle prototypes. This problem requires the determination of the variant of each prototype and the sequence prototype production. Tests must be allocated to prototypes with suitable components. Also, a feasible schedule must satisfy temporal and problem-specific constraints. Our objective is to minimize the peak demand for prototypes required for the testing process.

In Chapter 4, we formulate our complete scheduling problem using MILP. As a general purpose solving platform, MILP allows us to concentrate only on the modeling part and rely on the standard configuration in the MILP solver to find a solution. However, the scalability is the main obstacle since MILP cannot handle our large instances.

Moreover, we suggest the lower bound MILP model which considers two basic principles: selecting a set of variants to satisfy the component requirements and determining the number of variants certainly required over some important time intervals. Although the model neglects other consequences, its solution can be treated as a lower bound value of the entire problem. Also, simplifying the model allows us to solve the largest instance. We can further compare the obtained lower bound with a solution from any approach to measure the real optimality gap.

In Chapter 5, we further apply the CP approach to solve the problem. Unlike MILP, CP helps us to naturally express our complex set of requirements in a compact and scalable formulation. Moreover, we introduce our own search strategy trying to schedule critical tests first and use as few resources as possible. CP optimally solves the small problems and manages to generate good feasible solutions for our large real-life instances.

CP is quite convenient to develop a new application since most CP solvers provide some standard search algorithms. Users can concentrate on the modeling part only. The performance can be tweaked later using a more specific search. However, CP is quite inefficient to handle the optimization problem, especially for our objective function, as only poor lower bounds are obtained. Without the lower bounds from MILP, we cannot realize that CP has already achieved the feasible solutions which are almost optimal.

Moreover, we suggest trade-off procedures based on solving CP models. The analysis helps us to realize how much the makespan can be reduced or how many late tests can happen if we are allowed to build more or less prototypes than necessary.

In Chapter 6, we apply a hybrid approach based on Bender's decomposition using MILP and CP to iteratively address the planning and the scheduling problems. Also, we suggest several kinds of Bender's cut apart from the standard nogood constraint.

The hybrid approach can optimally solve a large instance if the characteristics of the instance are suitable for decomposition by our MILP model. Otherwise, no feasible solution is achieved during the process since this hybrid approach spends all efforts to increase the lower bound until it eventually obtains an optimal solution.

Our hybrid alternatives let the slave CP model correct the intermediate decision obtained from the master MILP model, while the stronger Bender's cuts can be introduced. CP can be more efficient to look for other feasible solutions without changing the objective function. However, the complexity of solving the slave problem increases. We can even lose the benefit of decomposition if the slave problem becomes too difficult for CP.

In Chapter 7, we further verify the performance and robustness of our methods by using the similar case study and its random instances. We slightly modify our models to consider different constraints. We show that CP still remains robust and efficient, even when using the basic search schemes. The hybrid approach, however, can only provide the lower bounds since the effect of several new constraints cannot be observed by the master problem.

More sophisticated branch schemes are also proposed to explore wider search regions and further improve our results. When comparing CP with the heuristic method, we find better feasible solutions for most of the instances and reduce the average number of required prototypes.

It takes CP several minutes of computation time to achieve its solutions while the heuristic method is clearly faster. However, our computation time is still within a reasonable range, especially for the long-term planning. Moreover, we need not change the solving algorithm after modifying the side constraints. Furthermore, the knowledge acquired from using the heuristic method can provide us with a guideline for developing an efficient search strategy in CP.

8.2 Future Research

Although our scheduling problem originally arises from the automotive industry, we believe that our solving techniques can be generally applied to any similar application dealing with the minimization of the resource demand in a parallel machine environment.

Therefore, it is still challenging to further improve our methods using our case study as a benchmark. We would like to suggest some promising research areas to be carried out in the future.

For the MILP approach, we can replace the Big-M formulations in the complete model by the indicator constraints which have recently been introduced in CPLEX [24]. The resulting formulations tend to be numerically more robust and accurate. However, we may still encounter a scalability problem.

Regarding the CP approach, we can improve its performance and robustness by using more advanced search frameworks. Instead of the standard depth-first search, we can also apply the limited discrepancy search which tries to follow a path suggested by a heuristic method as long as a number of wrong turns does not exceed a specified limit.

Also, we can use more intelligence backtracking policies, like backjumping, backmarking, or hybrid algorithms suggested by Prosser [49].

CP can be further combined with the local search methods. For instance, in ILOG Scheduler [26], CP is enhanced to cooperate with the large neighborhood search. We can shuffle between the resource allocation and job sequencing within the CP framework. We may specify a strategy to move tests out of a machine seldom used and insert them to other possible machines.

Concerning the MILP-CP hybrid approach, we may apply a Branch-and-Check framework suggested by Thorsteinsson [54]. The approach in our thesis tries to solve the master problem optimally before checking the overall feasibility. It can result in a large number of solving iterations if the optimal solution of the planning process is not realistic and quite far from the optimal solution for the whole schedule. The Branch-and-Check method suggests to check the feasibility of the slave problem whenever we find any integer feasible solution for the master problem. This may help us to find a better feasible solution faster.

Appendix A

Supplementary Computational Results

We provide the additional results obtained from solving the real-life and random instances. Remember that we obtain 4 real-life instances which contain 41, 100, 231, and 487 tests, respectively. Also, there are two sets of random instances: 100 instances of 20 tests and 60 instances of 600 tests.

A.1 Solving the Real-life Instances

We use the same footnotes as in Table A.1 for all tables in this section. The time limit is specified at 5 hours for the instance with 487 tests and at 1 hour for the rest. If the value of the total computation time exceeds the time limits, it means the computation terminates and achieves just a feasible solution without optimality proof.

Table A.1: Minimizing the number of required prototypes by CP with search algorithms in ILOG Scheduler

n	SelResMinGlobal			SelResMinLocal			SelAltRes		
	m_r	Time(s)	Total Time(s)	m_r	Time(s)	Total Time(s)	m_r	Time(s)	Total Time(s)
41	5[a]	0.13	0.14	5[a]	0.14	0.16	5[a]	0.05	0.07
100	5	5.08	3,609.44	5	5.42	3,610.32	5	0.51	3,606.15
231	21	6.42	3,610.52	22	6.68	3,610.70	23	6.27	3,610.83
487	121	1,398.36	18,012.40	121	1,425.12	18,104.40	-	-	-

[a] Optimal solution
Time: Computation time to achieve the solution
Total Time: Total computation time
-: No result within the time limit

Table A.2: Trade-off between the number of prototypes and the makespan for the instance with 231 tests using the SelResMinGlobal and SelResMin-Local alogrithms in ILOG Scheduler

m	SelResMinGlobal				SelResMinLocal			
	m_r	C_{max}	Time(s)	Total Time(s)	m_r	C_{max}	Time(s)	Total Time(s)
77	73	376[a]	47.23	47.57	70	376[a]	26.05	26.34
72	72	376[a]	33.49	33.76	no computation			
71	71	376[a]	31.04	31.23	no computation			
70	70	376[a]	23.19	23.38	no computation			
69	69	376[a]	23.36	23.53	69	376[a]	11.96	12.12
68	68	376[a]	21.30	21.48	68	376[a]	15.04	15.19
67	67	376[a]	22.65	22.86	67	376[a]	14.81	14.98
66	66	376[a]	19.86	20.02	66	376[a]	12.15	12.34
65	65	376[a]	18.95	19.13	65	376[a]	11.88	12.05
64	64	376[a]	18.21	18.35	64	376[a]	11.27	11.44
63	63	376[a]	18.27	18.42	63	376[a]	11.84	12.02
62	62	376[a]	15.18	15.33	62	376[a]	9.28	9.43
61	61	376[a]	13.46	13.63	61	376[a]	7.91	8.04
60	60	376[a]	13.59	13.74	60	376[a]	8.23	8.37
59	59	376[a]	10.61	10.78	59	376[a]	6.90	7.05
58	58	376[a]	8.27	8.42	58	376[a]	5.03	5.17
57	57	376[a]	6.12	6.26	57	376[a]	4.58	4.72
56	56	376[a]	5.53	5.68	56	376[a]	4.79	4.91
55	55	376[a]	5.55	5.69	55	376[a]	4.01	4.14
54	54	376[a]	5.29	5.44	54	376[a]	4.15	4.30
53	53	376[a]	4.88	5.04	53	376[a]	4.15	4.32
52	52	376[a]	4.64	4.77	52	376[a]	3.71	3.83
51	51	376[a]	4.13	4.25	51	376[a]	3.71	3.86
50	50	376[a]	4.98	5.11	50	376[a]	3.74	3.87
49	49	376[a]	3.97	4.09	49	376[a]	3.83	3.94
48	48	376[a]	4.20	4.35	48	376[a]	3.65	3.77
47	47	376[a]	3.47	3.61	47	376[a]	3.76	3.87
46	46	376[a]	3.78	3.88	46	376[a]	3.46	3.58
45	45	376[a]	3.47	3.56	45	376[a]	3.65	3.77
44	44	376[a]	3.62	3.72	44	398	2.35	3,605.28
43	43	376[a]	3.48	3.60	43	376[a]	3.81	3.91
42	42	376[a]	3.66	3.78	42	376[a]	3.33	3.43
41	41	376[a]	3.46	3.55	41	376[a]	3.38	3.48
40	40	376[a]	3.43	3.53	-	-	-	-
39	39	376[a]	3.33	3.43	-	-	-	-
38	38	376[a]	3.20	3.30	-	-	-	-
37	37	376[a]	3.04	3.14	-	-	-	-
36	36	376[a]	3.03	3.14	-	-	-	-
35	35	376[a]	3.00	3.10	-	-	-	-

Continued on Next Page…

m	SelResMinGlobal				SelResMinLocal			
	m_r	C_{max}	Time(s)	Total Time(s)	m_r	C_{max}	Time(s)	Total Time(s)
34	34	463	1.56	3,604.75	-	-	-	-
33	33	376[a]	2.71	2.79	-	-	-	-
32	32	376[a]	2.55	2.62	-	-	-	-
31	31	390	1.75	3,603.32	-	-	-	-
30	30	390	1.70	3,603.47	-	-	-	-
29	29	390	1.62	3,603.50	-	-	-	-
28	28	390	1.54	3,603.32	-	-	-	-
27	27	376[a]	2.33	2.42	-	-	-	-
26	26	376[a]	2.24	2.31	-	-	-	-
25	25	381	1.91	3,605.21	-	-	-	-
24	24	381	8.81	3,604.88	-	-	-	-
23	23	386	69.78	3,605.34	-	-	-	-

Table A.3: Trade-off between the number of prototypes and the makespan for the instance with 231 tests using the SelAltRes alogrithm in ILOG Scheduler

m	SelAltRes			
	m_r	$C_{\max}$	Time(s)	Total Time(s)
77	24	376[a]	5.82	6.05
23	23	379	1.98	3,605.47

Table A.4: Trade-off between the number of prototypes and the makespan for the instance with 487 tests using search algorithms in Scheduler (SelAltRes cannot find any solution.)

m	SelResMinGlobal				SelResMinLocal			
	m_r	C_{max}	Time(s)	Total Time(s)	m_r	C_{max}	Time(s)	Total Time(s)
162	162	381[a]	7,166.72	7,173.86	162	381[a]	6,392.91	6,399.84
161	161	381[a]	7,031.49	7,038.05	161	381[a]	6,276.13	6,282.45
160	160	381[a]	6,909.15	6,915.50	160	381[a]	6,045.92	6,051.90
159	159	381[a]	6,793.29	6,799.60	159	381[a]	5,999.57	6,005.64
158	158	381[a]	6,665.32	6,671.61	158	381[a]	5,920.28	5,926.34
157	157	381[a]	6,521.04	6,527.20	157	381[a]	5,797.30	5,802.93
156	156	381[a]	6,424.89	6,431.78	156	381[a]	5,717.49	5,723.62
155	155	381[a]	6,285.50	6,291.81	155	381[a]	5,587.78	5,594.06
154	154	381[a]	6,159.98	6,166.32	154	381[a]	5,457.85	5,463.42
153	153	381[a]	6,022.47	6,028.90	153	381[a]	5,318.80	5,324.13
152	152	381[a]	5,962.65	5,969.06	152	381[a]	5,241.34	5,247.15
151	151	381[a]	6,761.71	6,767.76	151	381[a]	5,137.73	5,142.99

Continued on Next Page...

m	SelResMinGlobal				SelResMinLocal			
	m_r	C_{max}	Time(s)	Total Time(s)	m_r	C_{max}	Time(s)	Total Time(s)
150	150	381[a]	6,619.86	6,626.15	150	381[a]	5,025.25	5,030.37
149	149	381[a]	6,478.35	6,484.71	149	381[a]	4,904.02	4,909.18
148	148	381[a]	6,319.54	6,325.50	148	381[a]	4,830.36	4,835.17
147	147	381[a]	6,202.97	6,209.37	147	381[a]	4,770.09	4,774.96
146	146	381[a]	6,090.43	6,096.71	146	381[a]	4,864.26	4,869.32
145	145	381[a]	5,917.14	5,923.48	145	381[a]	4,719.35	4,724.28
144	144	381[a]	5,765.03	5,770.57	144	381[a]	4,578.94	4,583.74
143	143	381[a]	5,637.12	5,642.51	143	381[a]	4,479.51	4,485.09
142	142	381[a]	5,552.67	5,558.79	142	381[a]	4,400.24	4,405.26
141	141	381[a]	5,478.85	5,484.36	141	381[a]	4,268.64	4,273.61
140	140	381[a]	5,357.20	5,362.72	140	381[a]	4,126.12	4,131.02
139	139	381[a]	5,234.58	5,240.04	139	381[a]	4,056.63	4,061.47
138	138	381[a]	5,145.41	5,150.76	138	381[a]	4,039.40	4,044.22
137	137	381[a]	5,057.50	5,062.54	137	381[a]	3,881.34	3,885.78
136	136	381[a]	4,973.13	4,978.20	136	381[a]	4,054.24	4,058.72
135	135	381[a]	4,885.90	4,891.11	135	381[a]	3,994.83	3,999.43
134	134	382	4,771.50	18,015.60	134	382	3,900.31	18,015.30
133	133	382	4,667.92	18,020.60	133	385	3,708.28	18,014.70
132	132	385	4,487.08	18,015.50	132	386	3,642.66	18,013.00
131	131	393	4,073.28	18,012.40	131	385	3,937.77	18,020.00
130	130	393	3,957.51	18,013.20	130	392	3,585.38	18,013.00
129	129	424	2,847.83	18,013.00	129	392	3,507.17	18,013.20
128	128	438	2,324.16	18,012.70	128	443	1,825.23	18,013.10
127	127	445	2,134.55	18,013.00	127	445	1,733.03	18,015.30
126	126	445	2,082.01	18,013.80	126	457	1,619.24	18,013.20
125	125	505	1,376.81	18,013.40	125	479	1,473.72	18,013.40
124	124	509	1,267.08	18,012.50	124	500	1,343.50	18,013.20
123	123	511	1,225.25	18,012.60	123	504	1,265.69	18,012.80
122	122	515	1,195.80	18,011.60	122	515	1,203.77	18,012.30
121	121	526	1,116.55	18,012.10	121	526	1,137.86	18,012.80
120	120	600	330.09	18,012.30	120	535	1,043.81	18,012.30

A.2 Solving the Random Instances

We provide the computational results of the random instances obtained from the CP and hybrid approaches. For CP we specify the time limit is specified at 1 hour for all random instances. For the hybrid approach we set the time limits of 5 hours for total computation, 10 minutes for the master MILP and the slave CP, and 1 minute for the sub-CP.

Table A.5: Computational results from solving 100 instances of 20 tests by CP and the hybrid approach.

No.	Solu-tion	CP		Hybrid						
		Slack/ MinId	LST/ MinId	MILP time	Dis time	Com time	Total time	#It	#Dinf	#CInf
1	8	0.76	0.63	0.07	0.01	0.17	0.25	6	0	5
2	7	0.37	0.36	0.09	0.00	1.00	1.09	7	0	6
3	6	0.32	0.32	0.47	0.00	0.38	0.85	20	0	19
4	7	0.43	0.74	0.35	0.00	0.46	0.81	18	0	17
5	7	0.58	0.36	0.26	0.01	0.28	0.55	13	0	12
6	5	0.09	0.13	0.05	0.00	0.04	0.09	5	0	4
7	6	0.24	0.22	0.08	0.00	0.08	0.16	6	0	5
8	7	0.37	0.19	0.37	0.00	0.10	0.47	17	0	16
9	6	0.12	0.59	0.08	0.00	0.07	0.15	6	0	5
10	7	1.50	2.02	0.54	0.00	0.51	1.05	21	0	20
11	8	0.67	1.18	0.40	0.00	3.49	3.89	17	0	16
12	7	0.95	4.40	0.37	0.00	2.89	3.26	19	0	18
13	6	0.67	0.65	0.07	0.00	0.09	0.16	6	0	5
14	8	1.23	1.36	1.23	0.00	0.98	2.21	42	0	41
15	7	0.39	0.47	0.26	0.00	0.21	0.47	15	0	14
16	6	0.26	0.31	0.13	0.02	0.15	0.30	8	0	7
17	9	8.23	26.11	3.83	0.02	46.47	50.32	103	0	102
18	8	16.71	9.28	0.24	0.01	22.45	22.70	16	1	14
19	6	0.16	0.33	0.08	0.00	0.06	0.14	7	0	6
20	8	6.21	51.83	0.31	0.01	60.23	60.55	17	0	16
21	6	0.49	0.76	0.78	0.00	19.07	19.85	33	0	32
22	6	0.13	0.20	0.00	0.00	0.00	0.00	1	0	0
23	8	3.05	5.07	1.92	0.00	26.34	28.26	54	1	52
24	5	0.15	0.16	0.06	0.00	0.02	0.08	5	0	4
25	7	0.82	0.77	0.29	0.01	5.01	5.31	19	0	18
26	8	34.49	46.26	0.84	0.01	18.39	19.24	36	0	35
27	6	0.13	0.12	0.07	0.00	0.14	0.21	5	0	4
28	6	0.11	0.10	0.11	0.00	0.04	0.15	8	0	7
29	6	0.24	0.40	0.33	0.00	0.06	0.39	8	0	7
30	6	0.16	0.51	0.25	0.01	0.28	0.54	13	0	12
31	8	0.71	0.66	0.24	0.00	0.87	1.11	12	0	11
32	7	3.30	5.74	0.11	0.00	0.70	0.81	9	0	8

Continued on Next Page...

Table A.5 – Continued

No.	Solu-tion	CP		Hybrid						
		Slack/ MinId	LST/ MinId	MILP time	Dis time	Com time	Total time	#It	#Dinf	#CInf
33	6	0.33	0.54	0.12	0.00	0.07	0.19	7	0	6
34	9	63.46	86.11	4.75	0.01	95.10	99.86	89	0	88
35	7	0.42	0.82	0.37	0.00	0.15	0.52	16	0	15
36	7	0.93	2.08	0.53	0.01	2.38	2.92	17	0	16
37	6	0.12	0.14	0.04	0.00	0.00	0.04	5	0	4
38	6	0.17	0.69	0.13	0.00	0.07	0.20	7	0	6
39	4	0.10	0.13	0.01	0.01	0.00	0.02	2	0	1
40	7	3.67	6.60	0.25	0.00	1.65	1.90	16	0	15
41	7	0.33	0.61	0.25	0.01	0.72	0.98	16	0	15
42	6	0.14	0.13	0.10	0.00	0.08	0.18	8	0	7
43	6	0.28	0.32	0.05	0.00	0.27	0.32	4	0	3
44	7	1.06	0.84	0.82	0.01	1.14	1.97	34	0	33
45	7	0.99	0.80	0.08	0.00	0.29	0.37	7	0	6
46	6	0.13	0.23	0.11	0.01	0.17	0.29	8	0	7
47	6	0.14	0.14	0.00	0.00	0.01	0.01	2	0	1
48	9	323.14	340.38	1.43	0.02	124.45	125.90	31	0	30
49	5	0.11	0.10	0.06	0.00	0.03	0.09	5	0	4
50	7	0.89	2.49	0.47	0.03	5.91	6.41	22	0	21
51	7	0.79	0.83	0.55	0.00	1.29	1.84	23	0	22
52	8	26.80	13.68	0.94	0.00	8.80	9.74	36	0	35
53	5	0.12	0.19	0.11	0.00	0.11	0.22	7	0	6
54	8	2.37	9.94	0.61	0.00	2.14	2.75	25	0	24
55	7	1.42	2.86	0.07	0.00	0.14	0.21	6	0	5
56	7	7.34	4.34	0.02	0.00	0.04	0.06	2	0	1
57	9	32.75	38.80	7.71	0.04	156.89	164.64	97	24	72
58	8	8.19	7.82	1.49	0.01	5.70	7.20	50	0	49
59	8	0.76	0.69	1.09	0.01	1.80	2.90	36	0	35
60	7	1.57	1.42	0.04	0.01	0.51	0.56	4	0	3
61	7	0.37	0.49	0.30	0.00	0.52	0.82	14	0	13
62	6	0.36	0.56	0.07	0.00	0.71	0.78	7	0	6
63	7	0.52	1.25	0.37	0.00	0.32	0.69	16	0	15
64	6	0.12	0.18	0.08	0.00	0.08	0.16	6	0	5
65	6	0.11	0.10	0.08	0.00	0.03	0.11	6	0	5
66	10	19.76	43.46	1.25	0.01	61.00	62.26	35	3	31
67	6	0.29	0.16	0.16	0.00	0.07	0.23	10	0	9
68	8	2.53	1.87	1.32	0.02	1.13	2.47	40	0	39
69	7	1.17	3.53	0.13	0.00	0.43	0.56	9	1	7
70	7	6.53	8.85	0.07	0.01	1.61	1.69	8	0	7
71	8	10.04	37.00	1.80	0.03	3.35	5.18	47	0	46
72	6	0.17	0.16	0.11	0.01	0.24	0.36	9	0	8

Continued on Next Page. . .

Table A.5 – Continued

No.	Solu-tion	CP		Hybrid						
		Slack/ MinId	LST/ MinId	MILP time	Dis time	Com time	Total time	#It	#Dinf	#CInf
73	7	1.19	8.84	0.31	0.00	0.34	0.65	16	0	15
74	5	0.09	0.09	0.07	0.00	0.02	0.09	5	0	4
75	7	0.71	0.45	0.04	0.00	0.05	0.09	4	0	3
76	6	2.10	2.09	0.06	0.00	0.30	0.36	7	0	6
77	7	1.27	0.49	0.29	0.02	0.40	0.71	16	0	15
78	7	0.77	1.39	0.35	0.00	0.37	0.72	17	0	16
79	7	2.59	1.93	0.30	0.01	0.76	1.07	16	0	15
80	8	2.90	8.68	0.91	0.03	13.26	14.20	34	0	33
81	6	0.40	0.52	0.07	0.00	0.06	0.13	5	0	4
82	6	0.25	0.25	0.06	0.00	0.11	0.17	8	0	7
83	10	208.41	196.99	6.45	0.01	307.43	313.89	97	0	96
84	5	0.15	0.19	0.01	0.00	0.01	0.02	2	0	1
85	6	0.22	0.26	0.10	0.01	0.47	0.58	7	0	6
86	8	3.21	3.37	1.26	0.00	9.21	10.47	45	0	44
87	7	10.02	4.47	0.04	0.00	19.06	19.10	7	0	6
88	7	0.12	0.11	0.12	0.00	0.05	0.17	8	0	7
89	8	10.25	6.98	0.29	0.00	21.12	21.41	18	0	17
90	6	0.14	0.21	0.08	0.02	0.06	0.16	6	0	5
91	6	0.16	0.23	0.11	0.01	0.16	0.28	7	0	6
92	7	0.89	0.89	0.36	0.01	2.61	2.98	19	0	18
93	6	0.32	0.81	0.18	0.01	0.02	0.21	11	0	10
94	6	0.16	0.15	0.05	0.00	0.16	0.21	4	0	3
95	6	0.12	0.13	0.06	0.00	0.07	0.13	6	0	5
96	6	0.20	0.22	0.02	0.00	0.02	0.04	3	0	2
97	5	0.08	0.10	0.04	0.01	0.07	0.12	5	0	4
98	6	0.20	0.31	0.13	0.00	0.94	1.07	9	0	8
99	5	0.08	0.10	0.02	0.00	0.01	0.03	3	0	2
100	7	0.58	1.46	0.06	0.00	0.26	0.32	7	0	6
Avg	6.77	8.52	10.25	0.54	0.01	10.68	11.22	17.53	0.30	16.23

Table A.6: Computational results from solving 60 instances of 600 tests by search algorithms in ILOG Scheduler, Slack/MinId, and LST/MinId

No.	Standard Search Strategies in ILOG Scheduler						Specified Search Strategies			
	SelAltRes		Global Slack		Local Slack		LST/MinId		Slack/MinId	
	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)
1	72	219.4	106	8,302.64	147	916.41	67	189.75	62	272.33
2	98	175.5	-	-	-	-	96	261.02	89	324.14
3	-	-	-	-	-	-	86	256.62	84	254.39

Continued on Next Page...

No.	Standard Search Strategies in ILOG Scheduler						Specified Search Strategies			
	SelAltRes		Global Slack		Local Slack		LST/MinId		Slack/MinId	
	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)
4	-	-	139	2,292.31	140	1956.16	-	-	70	287.81
5	-	-	-	-	-	-	72	233.36	68	232.27
6	93	171.5	130	3,204.24	122	4123.42	82	291.92	79	252.27
7	73	188.07	122	5,082.47	-	-	67	242.62	68	190.8
8	98	171.27	-	-	-	-	108	257.58	98	264.52
9	-	-	81	11,377.50	-	-	71	201.67	63	375.33
10	-	-	-	-	-	-	-	-	69	304.31
11	73	202.41	84	10,438.00	147	871.59	63	312.43	63	229.34
12	-	-	108	6,780.30	130	3822.06	72	295.16	69	241.31
13	73	191.18	145	1,188.90	-	-	67	431.87	66	341.3
14	91	202.15	110	6,793.97	117	6093.26	81	238.55	79	240.69
15	-	-	121	5,026.34	-	-	66	237.89	66	280.78
16	-	-	125	5,022.84	-	-	73	245.36	72	238.58
17	-	-	-	-	-	-	-	-	67	238.68
18	92	199.02	-	-	-	-	88	289.56	82	240.32
19	-	-	-	-	-	-	69	288.22	71	191.74
20	-	-	89	10,250.80	147	879.65	-	-	62	241.31
21	88	183.47	119	5,257.52	129	3899.68	84	248.15	78	243.81
22	-	-	145	1,101.09	145	1103.88	69	333.58	67	325.57
23	-	-	-	-	-	-	63	231.81	64	277.03
24	-	-	111	7,287.17	147	881.72	66	288.8	63	235.43
25	-	-	146	1,036.02	-	-	71	208.41	66	300.77
26	91	201.94	99	8,378.49	95	9127.31	88	180.45	83	247.27
27	76	201.57	87	10,152.30	-	-	69	246.46	65	246.17
28	-	-	147	864.89	-	-	66	197.57	-	-
29	92	190.44	147	752.53	115	6094.44	90	177.53	87	245.3
30	-	-	-	-	-	-	83	367.76	83	237.14
31	81	205.45	83	11,106.20	113	7362.19	72	281.51	-	-
32	-	-	145	1,152.22	-	-	73	247.69	71	243.17
33	-	-	84	10,574.80	147	899	67	221.41	64	262.67
34	87	188.05	145	1,143.62	-	-	78	237.28	73	232.34
35	-	-	150	179.37	-	-	71	232.66	68	364.8
36	-	-	131	5,202.74	137	2485.78	85	294.71	79	243.08
37	-	-	-	-	150	194.25	77	252.62	78	245.51
38	103	184.34	143	1,381.66	-	-	105	173.5	96	256.91
39	76	192.27	88	10,335.20	-	-	74	188.67	70	237.66
40	-	-	-	-	-	-	105	371.02	100	348.68
41	-	-	-	-	-	-	70	254.6	68	293.98
42	77	201.47	88	10,041.70	130	4142.34	68	296.36	68	259.21
43	-	-	87	9,719.69	-	-	68	243.47	62	360.44
44	-	-	133	3,452.93	150	215.28	66	193.89	64	200.22

Continued on Next Page...

No.	Standard Search Strategies in ILOG Scheduler						Specified Search Strategies			
	SelAltRes		Global Slack		Local Slack		LST/MinId		Slack/MinId	
	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)
45	85	176.33	137	2,415.44	-	-	83	265.07	82	257.79
46	-	-	-	-	-	-	66	243.08	-	-
47	-	-	-	-	-	-	81	179.03	78	236.85
48	75	196.98	-	-	-	-	66	290.41	64	233.07
49	88	173.74	113	5,780.20	131	2683.09	86	306.65	81	245.79
50	-	-	140	2,088.20	140	2119.71	-	-	75	260.57
51	-	-	140	1,856.29	123	4674.32	83	299.73	76	239.24
52	-	-	-	-	-	-	100	251.78	93	311.57
53	-	-	-	-	-	-	67	245.92	66	315.83
54	-	-	87	10,895.50	105	8732.95	64	813.03	66	257.28
55	-	-	-	-	-	-	70	355.96	66	432.74
56	-	-	-	-	-	-	108	261.11	100	259.51
57	-	-	85	10,474.50	146	1099.23	70	252.41	69	205.85
58	73	203.34	86	10,535.60	146	1058.63	65	252.33	63	322.94
59	-	-	-	-	-	-	97	250.22	92	314.59
60	78	202.66	89	10,016.20	147	856.25	70	550.26	-	-

Table A.7: Computational results from solving 60 instances of 600 tests by LST/MaxSt, Slack/MaxSt, LST-Slack1/MinId, LST-Slack2/MinId, and LST-Slack3/MinId

No.	LST/ MaxSt		Slack/ MaxSt		LST-Slack1/ MinId		LST-Slack2/ MinId		LST-Slack3/ MinId	
	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)
1	67	191.58	62	274.55	65	233.1	66	276.29	64	234.44
2	96	263.35	89	325.9	93	250.69	93	329.39	91	259.81
3	88	322.01	81	306.57	86	257.03	84	250.92	84	255.38
4	-	-	70	290.26	71	292.21	70	289.99	69	342.36
5	72	235.27	68	279.38	68	240.26	70	229.76	70	233.32
6	82	248.71	79	253.54	79	299.67	78	248.69	77	300.99
7	67	244.33	65	287.26	67	194.14	69	191.97	64	262.14
8	108	260.27	98	266.53	98	497.19	100	260.45	98	267.62
9	71	203	63	378.7	69	203.95	64	521.86	65	249.07
10	-	-	69	305.55	69	248.68	70	297.87	69	353.33
11	64	273.78	63	233.78	66	192.16	65	234.83	66	194.54
12	72	247.18	68	244.24	69	240.87	69	245.55	69	244.04
13	68	392.92	66	344.5	68	396.52	65	378.12	68	349.79
14	81	241.22	79	243.27	81	242.55	79	244.47	80	244.48
15	66	239.58	66	283.21	66	326.82	64	366.95	66	236.82

Continued on Next Page...

No.	LST/ MaxSt		Slack/ MaxSt		LST-Slack1/ MinId		LST-Slack2/ MinId		LST-Slack3/ MinId	
	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)
16	73	247.46	72	243.01	71	240.63	70	240.67	69	242.7
17	-	-	68	192.99	65	239.31	65	330.59	66	193.9
18	88	253.93	82	241.51	81	297.36	82	243.39	83	242.1
19	69	291.18	71	193.56	70	242.53	69	244.06	68	285.58
20	-	-	62	243.42	65	204.86	64	334.24	62	204.51
21	84	250.25	80	302.23	84	191.28	83	244.97	79	254.03
22	69	335.71	68	285.67	68	287.04	67	380.08	66	246.56
23	63	233.6	62	362.78	64	230.99	63	190.51	-	-
24	66	246.4	63	238.7	63	244.47	63	244.83	63	239.39
25	68	261.03	66	302.62	65	394.1	65	251.74	66	254.63
26	88	243.88	84	245.28	86	181.11	86	185.21	85	245.26
27	70	246.99	65	247.94	71	203.34	69	198.79	70	192.65
28	66	199.27	-	-	-	-	-	-	-	-
29	90	178.86	87	247.12	92	178.84	90	181.97	87	182.69
30	87	244.42	83	239.08	84	179.24	84	239.22	81	238.67
31	72	285.08	70	194.19	69	240.27	72	192.62	68	233.71
32	72	299.49	70	244.99	70	300.35	71	243.83	71	242.02
33	67	225.04	63	305.18	68	220.7	66	305.4	68	263.4
34	78	238.02	72	234.64	76	237.26	76	233.94	74	232.4
35	71	234.78	68	367.78	68	361.92	68	366.41	68	275.07
36	85	249.87	79	246.55	82	250.42	82	185	79	243.38
37	77	254.34	77	299.82	77	251.67	78	302.61	76	249.24
38	105	175.16	96	325.88	97	257.16	94	651.1	95	328.2
39	74	190.35	70	240.39	76	244.77	72	240.97	74	239.46
40	105	373.6	100	352.33	101	353.74	100	274.01	101	272.06
41	70	256.64	68	296.32	69	249.99	70	254.63	69	249.85
42	68	257.61	68	260.44	68	261.37	67	261.32	67	304.4
43	68	245.3	63	323.18	68	386	68	246.62	63	372.65
44	66	195.3	64	201.75	-	-	70	192.4	63	464.13
45	83	267.7	82	260.15	82	370.52	80	315.98	81	262.58
46	66	244.91	-	-	66	237.66	-	-	67	199.38
47	81	180.7	77	238.88	80	235.32	81	235.72	78	239.82
48	66	242.02	64	234.98	68	237.59	69	185.15	65	235.1
49	86	308.08	81	303.69	83	250.56	84	250.17	81	420.07
50	-	-	75	262.86	77	261.27	77	205.53	-	-
51	83	301.95	76	240.79	77	288.5	77	241.77	76	293.72
52	100	254.31	93	314.16	97	248.23	95	249.84	94	251.13
53	67	247.95	66	318.26	66	192.28	69	188.96	66	289.04
54	65	556.24	66	258.54	66	304.51	66	341.17	63	636.92
55	70	358.97	66	435.56	68	449.64	67	391.2	66	474.73
56	108	263.04	99	334.14	99	402.58	99	332.55	98	403.7

Continued on Next Page...

No.	LST/ MaxSt		Slack/ MaxSt		LST-Slack1/ MinId		LST-Slack2/ MinId		LST-Slack3/ MinId	
	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)
57	70	256.21	69	208.58	72	205.1	69	256.45	69	255.16
58	65	254.99	63	326.13	62	248.85	-	-	-	-
59	97	252.35	92	316.12	94	247.92	96	245.29	93	319.97
60	70	440.44	67	279.35	69	284.68	69	833.4	70	237

Table A.8: Computational results from solving 60 instances of 600 tests by LST-Slack1/ MaxSt, LST-Slack2/MaxSt, LST-Slack3/MaxSt, Slack-LST1/MinId, and Slack-LST2/MinId

No.	LST-Slack1/ MaxSt		LST-Slack2/ MaxSt		LST-Slack3/ MaxSt		Slack-LST1/ MinId		Slack-LST2/ MinId	
	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)
1	65	233.26	66	276.82	64	233.69	62	274.03	62	275.43
2	94	251.77	92	258.98	91	261.3	92	259.99	91	326.39
3	85	317.03	84	251.44	84	254.56	81	374.1	84	256.39
4	71	292.3	70	288.39	69	290.85	70	292.14	71	242.29
5	68	240.28	70	230.12	70	233.35	68	229.34	68	229.29
6	79	302.16	78	247.96	77	300.87	78	299.45	79	243.94
7	67	193.96	67	240.84	64	262.23	69	194.96	67	197.7
8	98	418.71	100	260.31	98	264.18	98	266.64	98	265.94
9	69	203.69	64	522.93	65	249.35	67	250.5	67	250.87
10	69	248.67	71	250.11	69	354.83	69	255.53	69	257.29
11	66	192.17	65	234.42	65	232.89	64	275.07	64	275.7
12	69	240.75	69	245.55	69	243.93	69	244.82	69	244.46
13	68	396.09	65	379.22	-	-	65	391.14	66	285.99
14	81	242.77	79	244.98	80	244.46	79	243.83	79	243.84
15	66	326.84	64	367.19	66	237.18	66	238.03	66	238.1
16	71	242.51	70	240.81	-	-	70	242.88	70	243.33
17	65	238.91	68	194.42	66	193.85	-	-	63	232.15
18	81	345.93	82	242.55	83	242	83	241.49	82	241.57
19	73	194.11	69	244.1	68	285.07	68	285.65	68	328.94
20	65	204.83	64	334.25	62	205.33	62	287.95	62	289.06
21	84	191.42	83	185.08	77	302.36	78	244.88	78	245.35
22	66	372.54	67	379.78	66	243.46	66	285.62	66	286.44
23	64	231.13	63	189.94	-	-	61	192.21	-	-
24	63	245.2	63	245.99	63	238.99	64	198.38	63	240.71
25	72	198.32	65	249.73	66	255.16	66	250.63	66	249.13
26	87	180.75	85	184.28	84	244.63	82	247.69	82	241.95
27	71	203.43	69	198.44	71	193.7	65	248.46	65	248.43

Continued on Next Page...

No.	LST-Slack1/ MaxSt		LST-Slack2/ MaxSt		LST-Slack3/ MaxSt		Slack-LST1/ MinId		Slack-LST2/ MinId	
	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)
28	-	-	-	-	-	-	64	246.78	64	249.42
29	90	247.26	90	181.66	87	184.24	88	244.25	88	243.84
30	84	179.65	85	240.21	81	300.08	82	238.08	81	300.27
31	69	240.32	72	192.25	68	234.38	71	192.87	69	282.25
32	70	301.24	70	247.15	70	243.61	70	249.07	71	249.36
33	68	220.71	66	304.73	68	262.84	65	265.94	65	258.83
34	77	237.77	76	234.25	73	233.56	73	236.02	73	232.73
35	68	361.03	68	365.71	68	274.78	69	235.37	68	279.83
36	82	250.24	82	184.98	79	243.99	79	298.33	80	244.1
37	77	252.04	77	302.59	76	248.41	78	246.92	77	244.82
38	97	257.88	95	325.29	95	330.26	96	261.91	96	336.3
39	76	244.79	72	241.1	74	239.66	74	243.12	73	191.51
40	101	353.64	100	273.07	101	273.66	100	350.67	100	352.9
41	68	250.34	70	254.33	69	254.03	67	392.91	69	345.76
42	68	260.93	69	262.01	69	261.28	69	211.51	69	211.67
43	69	339.71	67	294.39	63	372.94	64	328.48	64	327.06
44	-	-	70	192.17	63	464.36	66	194.16	62	292.9
45	82	372.08	80	316.27	81	262.93	83	259.31	82	258.41
46	66	238.36	-	-	67	198.61	-	-	66	199.27
47	80	234.84	82	179.44	78	239.74	76	233.6	76	232.39
48	68	238.72	69	185.2	65	234.55	-	-	65	236.95
49	83	249.14	84	250.95	81	419.86	83	247.27	80	308.48
50	77	261.4	77	206.4	-	-	74	256.61	73	308.91
51	77	288.57	77	286.72	76	293.56	77	237.18	76	236.93
52	97	248.28	95	249.47	94	251.25	94	245.31	97	171.95
53	66	191.74	69	189.73	66	289.22	68	246.34	71	193.75
54	66	301.81	69	198.91	68	459.99	71	205.44	64	303.03
55	68	447.76	67	389.87	66	474	68	347.15	66	436.63
56	100	333.6	99	405.54	98	403.79	100	261.38	98	398.15
57	71	252.07	69	256.05	72	205.7	69	254.29	69	259.1
58	62	249.56	65	244.52	69	201.42	63	371.61	64	200.67
59	94	248.34	95	247.05	93	322.95	92	321.19	92	249.29
60	69	284.16	71	238.8	70	238	67	235.25	-	-

Table A.9: Computational results from solving 60 instances of 600 tests by Slack-LST3/ MinId, Slack-LST1/MaxSt, Slack-LST2/MaxSt, and Slack-LST3/MaxSt

No.	Slack-LST3/ MinId		Slack-LST1/ MaxSt		Slack-LST2/ MaxSt		Slack-LST3/ MaxSt	
	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)
1	62	274.73	62	274.93	62	275.66	62	274.8
2	92	261.96	92	259.84	89	454.01	92	262.26
3	83	255.41	81	307.44	84	255.72	83	255.74
4	71	238.63	70	243.65	71	242.49	71	238.55
5	68	228.75	68	229.1	68	228.87	68	228.66
6	78	251.52	77	246.15	77	294.4	77	300.35
7	63	247.31	69	195.42	63	292.35	63	247.27
8	98	345.25	98	267.15	98	266.03	98	346.22
9	67	249.97	67	250.58	67	250.68	67	249.81
10	69	259.02	69	255.7	69	257.01	69	258.91
11	64	276.84	64	273.46	64	275.48	64	275.93
12	68	287.94	68	244.96	68	244.32	67	336.29
13	66	285.85	70	195.93	66	285.8	65	352.76
14	79	244.16	79	243.72	79	243.81	79	243.88
15	66	284.54	66	238.02	66	238.11	66	286.3
16	70	241.08	70	242.56	70	242.34	70	241.06
17	63	327.76	-	-	63	232.06	63	327.36
18	83	242.82	83	242.4	82	242.42	83	244.22
19	68	280.77	68	285.66	68	284.82	68	280.22
20	62	289.11	62	287.27	62	289.84	62	289.67
21	78	243.71	78	244.64	78	297.93	78	244.44
22	66	281.53	67	244.66	67	244.13	66	281.6
23	67	190.82	61	192.53	-	-	67	190.7
24	65	199.45	64	199.83	63	241.65	63	240.47
25	68	202.79	66	300.82	66	299.86	66	247.58
26	82	245.06	83	249.86	83	241.76	83	245.31
27	68	245.17	65	247.94	65	247.76	67	288.25
28	64	249.41	64	246.79	64	248.61	64	249.26
29	87	247.35	88	244.51	88	244.19	87	247.06
30	82	238.94	82	239.09	81	300.11	82	238.1
31	71	195.13	69	237.96	70	194.83	71	195.13
32	71	197.13	69	248.76	69	248.35	71	197.51
33	63	257.56	65	264.35	66	216.5	-	-
34	74	235.52	72	234.33	72	235.7	74	294.91
35	69	275	69	235.84	68	280.55	69	273.75
36	80	245.93	79	297.2	80	243.54	80	246.1
37	77	245.48	77	300.65	77	298.09	77	298.06
38	96	331.83	96	330.77	96	335.96	96	336.63

Continued on Next Page...

No.	Slack-LST3/ MinId		Slack-LST1/ MaxSt		Slack-LST2/ MaxSt		Slack-LST3/ MaxSt	
	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)	Sol.	Time(s)
39	73	191.76	74	243.11	73	191.56	73	191.52
40	98	344.31	100	351.41	100	354.89	99	267.77
41	70	201.68	73	200.4	67	342.84	68	299.29
42	67	305.51	68	259.29	68	258.92	68	259.01
43	64	374.22	64	328.01	64	329.08	64	374.31
44	-	-	66	194.49	63	248.26	62	199.27
45	82	315.52	83	258.38	82	258.88	82	315.61
46	65	199.72	-	-	66	199.38	65	201.56
47	76	236.85	75	233.97	75	232.59	76	236.46
48	67	230.66	-	-	65	236.41	67	231.15
49	82	251.14	83	247.62	80	308.36	82	250.69
50	73	263.26	74	255.42	73	308.31	73	263.32
51	76	236.5	77	237.21	76	237.66	76	236.27
52	94	245.13	94	244.67	97	171.75	94	244.82
53	65	292.69	-	-	71	193.09	65	292.76
54	65	401.79	71	205.52	65	260.18	-	-
55	66	440.49	68	347.35	66	437.56	66	440.91
56	99	400.12	99	333	99	332.85	99	401.73
57	69	253.29	69	253.96	72	210.46	68	255.03
58	64	200.9	63	371.16	64	200.72	64	200.33
59	92	247.85	92	247.71	92	250.48	92	250.34
60	-	-	67	235.55	68	236.28	69	281.15

Table A.10: Computational results from solving 60 instances of 600 tests by Backward Planning/MinId and Backward Planning/MaxSt

No.	Backward/ MinId		Backward/ MaxSt		No.	Backward/ MinId		Backward/ MaxSt	
	Sol.	Time(s)	Sol.	Time(s)		Sol.	Time(s)	Sol.	Time(s)
1	-	-	-	-	31	69	45.49	69	43.88
2	-	-	-	-	32	-	-	-	-
3	-	-	-	-	33	-	-	-	-
4	-	-	-	-	34	-	-	-	-
5	-	-	-	-	35	-	-	-	-
6	-	-	-	-	36	-	-	-	-
7	-	-	-	-	37	74	52.02	74	51.69
8	96	144.47	-	-	38	-	-	-	-
9	-	-	-	-	39	-	-	-	-
10	67	88.92	66	134.86	40	-	-	-	-
11	62	36.86	-	-	41	-	-	-	-
12	66	43.8	66	44.02	42	-	-	-	-
13	-	-	-	-	43	-	-	-	-
14	78	48.24	78	48.6	44	-	-	-	-
15	65	39.01	65	39.19	45	81	51.85	81	52.49
16	70	47.5	70	47.62	46	-	-	-	-
17	-	-	-	-	47	-	-	77	50.21
18	-	-	-	-	48	-	-	-	-
19	-	-	-	-	49	-	-	-	-
20	-	-	-	-	50	-	-	-	-
21	77	50.07	-	-	51	74	91.23	74	91.2
22	-	-	-	-	52	92	65.81	92	66.42
23	-	-	-	-	53	64	81.67	64	82.01
24	61	71.49	61	72.08	54	63	125.05	63	127.35
25	-	-	-	-	55	-	-	-	-
26	81	55.59	81	56.17	56	-	-	-	-
27	-	-	-	-	57	-	-	-	-
28	-	-	-	-	58	-	-	-	-
29	-	-	-	-	59	-	-	-	-
30	81	55.3	81	109.57	60	-	-	-	-

Bibliography

[1] Emile Aarts and Jan Karel Lenstra, editors. *Local Search in Combinatorial Optimization.* Wiley-Interscience publication, 1997.

[2] Ionut D. Aron, John Hooker, and Tallys H. Yunes. SIMPL: A system for integrating optimization techniques. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3011 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2004.

[3] Tapan P. Bagchi. *Multiobjective Scheduling by Genetic Algorithms.* Kluwer Academic Publishers, 1999.

[4] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: applying constraint programming to scheduling problems.* Kluwer, 2001.

[5] Roman Barták. *Intelligent Manufacturing Systems 2003*, chapter Constraint-based Scheduling: An Introduction for Newcomers, pages 69–74. Elsevier Science, 2003.

[6] Jan-Hendrik Bartels and Jürgen Zimmermann. Scheduling tests in automotive r&d projects. *European Journal of Operational Research*, 193(3):805–819, 2009.

[7] Nicolas Beldiceanu, Helmut Simonis, Philip Kay, and Peter Chan. The CHIP system. White Paper COSY/WHITE/002, Cosytec SA, 1997. http://www.cosytec.com.

[8] Luca Benini, Davide Bertozzi, Alessio Guerri, and Michela Milano. Allocation and scheduling for MPSoCs via decomposition and no-good generation. In *Principles and Practice of Constraint Programming — CP 2005*, volume 3709/2005 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2005.

[9] Robert E. Bixby, Mary Fenelon, Zongao Gu, Ed Rothberg, and Roland Wunderling. MIP: Theory and practice — closing the gap. In *Systems modelling and optimization: Methods, theory, and applications*, volume 174, pages 19–49. Kluwer, 2000.

[10] Alexander Bockmayr and Nicolai Pisaruk. Detecting infeasibility and generating cuts for mixed integer programming using constraint programming. *Computers & Operations Research*, 33(10):2777–2786, 2006.

[11] Alberto Caprara, Paolo Toth, and Matteo Fischetti. Algorithms for the set covering problem. *Annals of Operations Research*, 98:353–371, 2000.

[12] Massimiliano Caramia. *Effective resource management in manufacturing systems : optimization algorithms for production planning.* Springer, 2006.

[13] Jacques Carliear and Eric Pinson. A practical use of jackson's preemptive schedule for solving the job-shop problem. *Annals of Operations Research*, 26:269–287, 1990.

[14] Andrew M. Cheadle, Warwick Harvey, Andrew J. Sadler, Joachim Schimpf, Kish Shen, and Mark G. Wallace. ECLiPSe: A tutorial introduction. Technical Report IC-Parc-03-1, IC-Parc, Imperial College London, 2003.

[15] Emilie Danna and Claude Le Pape. *Constraint and Integer Programming: Toward a Unified Methodology*, chapter Two generic schemes for efficient and robust cooperative algorithms, pages 33–58. Kluwer, 2004.

[16] Andrew Davenport, Jayant Kalagnanam, Chandra Reddy, Stuart Siegel, and John Hou. An application of constraint programming to generating detailed operations schedules for steel manufacturing. In *Principles and Practice of Constraint Programming: CP 2007*, volume 4741/2007 of *Lecture Notes in Computer Science*, pages 64–76. Springer, 2007.

[17] Erik Demeulemeester. Minimizing resource availability costs in time-limited project networks. *Management Science*, 41(10):1590 – 1599, 1995.

[18] Martin E. Dyer and Laurence A. Wolsey. Formulating the single machine sequencing problem with release dates as a mixed integer program. *Discrete Applied Mathematics*, 26:225–270, 1990.

[19] Kelly Easton, George Nemhauser, and Miachael Trick. *Constraint and Integer Programming: Toward a Unified Methodology*, chapter CP Based Branch-and-Price, pages 207–231. Kluwer, 2004.

[20] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1999.

[21] John Hooker. Planning and scheduling to minimize tardiness. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*, pages 314–327. Springer, 2005.

[22] John Hooker. An integrated method for planning and scheduling to minimize tardiness. *Constraints*, 11:139–157, July 2006.

[23] Chih C. Hsu and David S. Kim. A new heuristic for the multi-mode resource investment problem. *Journal of the Operational Research Society*, 56:406–413, 2005.

[24] ILOG CPLEX. *ILOG CPLEX 10.0 User's Manual.* ILOG S.A., January 2006.

[25] ILOG OPL. *ILOG OPL Studio 3.7 Language Manual.* ILOG S.A., September 2003.

[26] ILOG Scheduler. *ILOG Scheduler 6.2 Reference Manual.* ILOG S.A., January 2006.

[27] ILOG Solver. *ILOG Solver 6.2 Reference Manual.* ILOG S.A., January 2006.

[28] *Diagnosing Memory in ILOG CPLEX Applications.* ILOG S.A., 2008. `http://www.ilog.com/products/optimization/info/memory.cfm`.

[29] Vipul Jain and Ignacio E. Grossmann. Algorithms for hybrid MILP/CP models for a class of optimization problems. *INFORMS Journal on Computing*, 13:258–276, 2001.

[30] Ulrich Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence (IJCAI 01)*, 2001.

[31] Sachin Karadgi. Scheduling of tests on vehicle prototypes. Master's thesis, Dortmund University, 2005.

[32] Stephan Kohlhoff. *Product Development with SAP in the Automotive Industry.* SAP Press, 2007.

[33] Philippe Laborie. Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new resutls. *Artificial Intelligence*, 143:151–188, 2003.

[34] Claude Le Pape. Constraint-based scheduling: A tutorial. First International Summer School on Constraint Programming, 2005.

[35] Claude Le Pape, Philippe Couronné, Didier Vergamini, and Vincent Gosselin. Time-versus-capacity compromises in project scheduling. In *Proceedings of the Thirteenth Workshop of the U.K. Planning Special Interest Group*, 1994.

[36] Kamol Limtanyakul. Scheduling of tests on vehicle prototypes using constraint and integer programming. In *Operations Research Proceedings 2007*, pages 421–426, Saarbrücken, 2007. Springer.

[37] Kamol Limtanyakul and Uwe Schwiegelshohn. Investigation on the use of hybrid approach for scheduling of tests on vehicle prototypes. In *Proceedings of the Doctoral Programme of the Thirteenth International Conference on Principles and Practice of Constraint Programming*, pages 67–72, Providence, September 2007.

[38] Kamol Limtanyakul and Uwe Schwiegelshohn. Scheduling tests on vehicle prototypes using constraint programming. In *Proceedings of the 3rd Multidisciplinary International Scheduling Conference: Theory and Applications*, pages 336–343, Paris, August 2007.

[39] Jeffrey T. Linderoth and Ted K. Ralphs. *Integer Programming: Theory and Practice*, chapter Noncommercial Software for Mixed-Integer Linear Programming, pages 253–305. CRC Press, 2006.

[40] Jeffrey Lockledge, Dimitrios Mihailidis, John Sidelko, and Kenneth Chelst. Prototype fleet optimization model. *Journal of the Operational Research Society*, 53:833–841, 2002.

[41] Fae Martin, Arthur Pinkney, and Xinghuo Yu. Cane railway scheduling via constraint logic programming: Labelling order and constraints in a real-life application. *Annals of Operations Research*, 108:193–209, 2001.

[42] Michela Milano and Michael Trick. *Constraint and Integer Programming: Toward a Unified Methodology*, chapter Constraint and Integer Programming: Basic Concepts, pages 1–28. Kluwer, 2004.

[43] Rolf H. Möhring. Minimizing costs of resource requirements in project networks subject to a fixed completion time. *Operations Research*, 32(1):89 – 120, 1984.

[44] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, 1999.

[45] Klaus Neumann and Jürgen Zimmermann. Resource levelling for projects with schedule-dependent time windows. *European Journal of Operational Research*, 117:591–605, 1999.

[46] Hartwig Nübel. A branch-and-bound procedure for the resource investment problem with generalized precedence constraints. Technical report, Institut für Wirtschaftstheorie und Operations Research, Universität Karlsruhe, 1998.

[47] Wim Nuijten and Claude Le Pape. Constraint-based job shop scheduling with ILOG Scheduler. *Journal of Heuristics*, 3(4):271–286, 1998.

[48] Michael Pinedo. *Scheduling-Theory, Algorithm and Systems*. Prentice Hall, 2nd edition, 2002.

[49] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3), 1993.

[50] Jean-Francois Puget and Irvin Lustig. Constraint programming and maths programming. In *The Knowledge Engineering Review*, volume 16, pages 5–23. Cambridge University Press, 2001.

[51] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[52] Ruslan Sadykov and Laurence A. Wolsey. Integer programming and constraint programming in solving a multi-machine assignment scheduling problem with deadlines and release dates. *INFORMS Journal on Computing*, 18:209–217, 2006.

[53] Robert Scheffermann, Uwe Clausen, and Andrea Preusser. Test-scheduling on vehicle prototypes in the automative industry. In *Proceedings of Sixth Asia-Pacific Industrial Engineering and Management Systems (APIEMS)*, volume 11, pages 1817–1830, Manila, 2005.

[54] Erlendur S. Thorsteinsson. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming. In *Principles and Pratice of Constraint Programming (CP2001)*, volume 2239 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2001.

[55] Pascal Van Hentenryck, Laurent Perron, and Jean-Francois Puget. Search and strategies in opl. *ACM Transactions on Computational Logic*, 1(2):285–320, October 2000.

[56] H. Paul Williams. *Model Building in Mathematical Programming*. Wiley, 2005.

[57] Laurence A. Wolsey. *Integer Programming*. Wiley-Interscience, 1998.

[58] Denie S. Yamashita, Vinícius A. Armentano, and Manuel Laguna. Scatter search for project scheduling with resource availability cost. *European Jounal of Operational Research*, 169:623–637, 2006.

[59] Armen Zakarian. Performance analysis of product validation and test plans. Annual progress report, Center for Engineering Education and Practice, College of Engineering and Computer Science, University of Michigan-Dearborn, 2000.