Constraints (2012) 17:172–203
DOI 10.1007/s10601-012-9118-y

APPLICATION

Improvements of constraint programming and hybrid methods for scheduling of tests on vehicle prototypes

Kamol Limtanyakul · Uwe Schwiegelshohn

Published online: 17 March 2012

Abstract In the automotive industry, a manufacturer must perform several hundreds of tests on prototypes of a vehicle before starting its mass production. Tests must be allocated to suitable prototypes and ordered to satisfy temporal constraints and various kinds of test dependencies. The manufacturer aims to minimize the number of prototypes required. We present improvements of constraint programming (CP) and hybrid approaches to effectively solve random instances from an existing benchmark. CP mostly achieves better solutions than the previous heuristic technique and genetic algorithm. We also provide customized search schemes to enhance the performance of general search algorithms. The hybrid approach applies mixed integer linear programming (MILP) to solve the planning part and CP to find the complete schedule. We consider several logical principles such that the MILP model can accurately estimate the prototype demand, while its size particularly for large instances does not exceed memory capacity. Moreover, the robustness is alleviated when we allow CP to partially change the allocation obtained from the MILP model. The hybrid method can contribute to optimal solutions in some instances.

Keywords Automotive industry · Constraint programming · Mixed integer linear programming · Benders decomposition

K. Limtanyakul (✉)
Department of Software Systems Engineering, The Sirindhorn International
Thai-German Graduate School of Engineering,
King Mongkut's University of Technology North Bangkok, Bangkok, Thailand
e-mail: kamoll@kmutnb.ac.th

U. Schwiegelshohn
Robotics Research Institute, Technische Universität Dortmund, 44227 Dortmund, Germany
e-mail: uwe.schwiegelshohn@udo.edu

1 Introduction

During the development process of new vehicle series, a car manufacturer must perform hundreds of tests to ensure that the products meet safety and quality standards. The mass production can start only after all tests are accomplished. As a suitable production does not exist yet these prototypes are handmade and expensive (between 0.5 to 1.5 million euro 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 a given prototype. Ignoring cost differences between various 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 the 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 suitable for processing 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 an appropriate variant.

Furthermore, we must consider various temporal restrictions that can be modeled as release and due dates. For instance, consider a driving test in winter 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.

Most companies have a special shop to produce the prototypes. Due to the limited capacity of this shop, the prototypes are sequentially manufactured resulting in an availability time for each prototype. Hence, the scheduler must ensure that tests can start only when the assigned prototype is constructed and sufficiently prepared for the test. Also, a valid schedule must obey additional constraints. For instance, a brake system test must be completed before starting a long-run test.

Therefore, our optimization problem has to determine a minimum number of prototypes required to complete all tests. Also, we have to consider the prototype variants such that tests can be allocated according to their component requirements. Finally, the start of tests must be determined in order to satisfy the temporal constraints and to ensure the sequence conditions like for the crash tests.

At the beginning, we applied constraint programming (CP) to solve the problem given by Scheffermann et al. [23]. Our CP formulation and computation results from a few real-life instances were reported in Limtanyakul and Schwiegelshohn [15]. We relied on a default search algorithm in our CP solver and obtained feasible solutions within a reasonable time.

However, it was difficult to justify the quality of the solutions obtained from CP. For the largest real-life instance, our CP feasible solution required 111 prototypes for processing all 487 tests. CP could prove the same problem infeasible if we set a limit of 9 prototypes available for testing. There was a large unknown optimality gap between 10 and 110 prototypes for which the CP approach failed to prove feasibility or infeasibility.

Afterward, we provided a preliminary study on the use of MILP-CP hybrid approach based on Benders decomposition [14]. To apply the hybrid approach, we formulated a planning MILP model using set-covering and energetic reasoning principles to estimate the prototype demand and resource allocation, while CP solved the rest of the scheduling process. The *nogood* constraint is applied to remove invalid solutions. We achieved an optimal solution needing 110 prototypes for the instance with 487 tests.

In this paper, we further improve our CP and hybrid methods to deal with the case given by Bartels and Zimmermann [3]. The new problem becomes larger with up to 600 tests and contains different additional constraints. Moreover, random instances were generated based on the characteristics of their real-life instance and suggestions from their industrial partner. The website[1] provides these instances together with their best solutions solved by heuristics and genetic algorithm (GA).

We adopted this benchmark to compare the performance and robustness of our methods. However, this problem cannot be efficiently solved when we only modify the mathematical models. It becomes essential to develop our own branching schemes for the CP approach. In principle, our branching schemes try to determine and allocate critical tests using several criteria like slack times or latest start times. Our new computation results here are much better than relying on the search algorithms provided in our CP solver.

In addition, we need to modify the hybrid method in order to solve the new problem. Our original hybrid method no longer achieved any solution [14]. For one of the instances, MILP kept the estimation of the demand at 66 prototypes, while Bartels and Zimmermann [3] found the best feasible solution which required 99 prototypes. We realized that the MILP model could not precisely estimate the prototype demand since we did not include the impact of some side constraints adding to the new problem. In this paper, we include a procedure suggested by Bartels [2] to determine an initial lower bound from solving a maximum clique problem. It helps us to improve both demand estimation and resource allocation for the planning of the MILP model.

Moreover, we introduce scheduling CP models which not only look for a complete solution, but also can partially alternate decisions from the planning process. As a result, the method becomes more robust, while stronger Benders constraints can be applied. Finally, we need to consecutively apply several branching schemes to solve each CP scheduling problem. This is essential for dealing with large instances as CP can effectively search in various potential regions. Due to these improvements our hybrid method contributes to new solutions with optimality proof.

The paper outline is as follows. Section 2 provides background information about previous work and some relevant projects. We further formally define our problem and introduce appropriate notations in Section 3. Section 4 provides the CP formulation and search strategy. For the hybrid approach in Section 5, we first describe mathematical principles to formulate the planning MILP model. Afterward, we suggest variants of the scheduling CP model. Section 6 shows the computational results, before we finally provide the conclusion.

[1] http://www.wiwi.tu-clausthal.de/testsets-evt

2 Literature survey

There are several projects related to the vehicle test process. Although car manufacturers have slight differences in problem characteristics and solving techniques, they all aim to reduce the cost for the production of prototypes.

Lockledge et al. [16] applied a multi-stage mathematical programming model to optimize the fleet of prototypes for the Ford Motor Company. In the first stage, they formulated a model based on the set covering principle to determine the number of required variants subject to component requirements of all tests. The second model determines the minimum number of prototypes 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 formally described in Section 3. Since we have to deal with so many distinct values of due dates, a large number of time slots and corresponding integer variables are required. Moreover, we have to consider other temporal constraints like release dates of tests and available times of prototypes. Consequently, the complexity of the MILP model quickly increases beyond memory capacity [14].

Scheffermann et al. [23] considered a problem with a number of release and due dates, including some additional conditions. For instance, two long-run tests cannot be executed on the same prototype. Also, a brake system test and a suspension system test must be carried out on the same prototype in order to verify the coordination of both systems. Scheffermann et al. [23] suggested a heuristic approach based on specific problem knowledge. Statistical methods were also applied to adjust parameters inside the algorithm. However, it is still difficult to achieve a parameter set which can efficiently reduce a number of required prototypes. This approach has been used as a decision support tool to help the planning department in a real working environment.

Bartels and Zimmermann [3] dealt with another similar problem as few side constraints are different. Instead of tests being executed on the same or on different prototypes, they considered partially ordered destructive tests like a driving test that can damage the chassis such that this prototype is not suitable for processing an acoustic test. Therefore, either this driving test is executed after the acoustic test or they are allocated on different prototypes. They presented a MILP formulation which can be applied to solve only small size problems. To handle larger instances, they proposed an heuristic approach based on a priority-rule. Afterward, GA method was applied to further reduce the number of required prototypes. Bartels [2] thoroughly described the implementation of his GA method.

Moreover, Zakarian [27] developed an analysis model and a decision support tool to evaluate the performance of product validation and test plans for General Motors Truck Group. His work concentrated on stochastic scheduling. He modeled uncertainties associated with the 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 the 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 simulation techniques.

In the standard scheduling framework given in Pinedo [22], we can consider tests and prototypes as jobs and machines, respectively. Also, the number of machines or resources is typically specified before solving the problems. However, our problem has to simultaneously consider both resource planning and scheduling. This problem is related to the resource investment problem (RIP) known as the problem of scarce time. Resources with unlimited capacity are initially given. We want to reduce the peak demand of resource utilization, while being able to finish all jobs within an overall project deadline. Note that we mostly use the terms like tests and prototypes throughout this paper. Jobs and machines are mentioned when we discuss theories or problems based on the standard framework.

Möhring [17] and Demeulemeester [5] presented exact algorithms which are applicable for solving instances with 20 jobs. Yamashita et al. [26] proposed a scatter search procedure together with multi-start heuristics to deal with larger problems of up to 120 jobs. Also, Neumann and Zimmermann [18] suggested a heuristic algorithm for a resource leveling problem that considers RIP as a special case.

In addition, Nübel [20] studied a problem RIP/max which included the minimum and maximum time lags between the start time of activities. He applied a depth-first branch-and-bound algorithm to explore a search scheme generated from pseudo-semiactive schedules. Hsu and Kim [7] suggested a priority rule heuristic approach for the multi-mode resource investment problem (MMRIP) in which each activity can be carried out by one of several alternative resources.

Finally, the MILP-CP method based on Benders decomposition originally developed by Jain and Grossmann [10] suggested that MILP and CP techniques have complementary strengths for solving a class of problems where only a subset of the binary variables have non-zero objective function coefficients when modelled as an MILP.

After that, this hybrid method was applied to solve various optimization problems [4, 6, 13]. Their hybrid methods were more efficient than using either MILP or CP when tested by instances having around 30 jobs.

3 Problem description and notations

After informally explaining our problem in Section 1, we introduce the notations used in the rest of the paper. These notations are based on those in the book of Pinedo [22] whenever possible.

$V = \{1, 2, ..., l\}$, $I = \{1, 2, ..., m\}$, and $J = \{1, 2, ..., n\}$ are the sets of prototype variants, prototypes, and tests, respectively. Prototype $i \in I$ belongs to variant $v_i \in V$. $M_j \subseteq V$ is the set of prototype variants that can perform test $j \in J$. Similarly, we define $N_v = \{j \in J \mid v \in M_j\}$ to represent the set of tests which can be allocated to prototype variant $v \in V$.

Each test $j \in J$ must be executed without interruption and has processing time p_j. All tests must be finished within an overall project deadline $\bar{d}$. We further include two dummy tests 0 and $n + 1$ with $p_0 = p_{n+1} = 0$ to be carried out at time 0 and at the deadline, respectively.

Moreover, the notation t_j represents the start time of test j which is unknown before solving the scheduling problem. There exits temporal constraints between start times of two different tests j and k which must be greater or equal to given

minimum time lags, i.e. $t_j - t_k \geq \delta_{kj}$. It is important to tighten the processing time windows of tests. We can solve a linear program of the problem considering only these temporal constraints, as shown by Li and Womer [13]. Alternatively, we modify the Floyd–Warshall algorithm with a time complexity of $O(n^3)$ to determine the distance matrix as we can later analyze which tests cannot be certainly executed on the same prototype, as shown in Section 5.1.

Following Neumann et al. [19], the network graph G contains $n+2$ nodes to represent n tests together with the dummy tests 0 and $n+1$. On the edge between nodes j and k, a weight of δ_{jk} is initially given. We compare δ_{jk} with the sum of weights $\delta_{jl} + \delta_{lk}$, where node l is any intermediate point to connect nodes j and k. After the consideration of all possible intermediate nodes, the longest path Δ_{jk} is selected and represented in the distance matrix. Note that this algorithm is a slight modification of the Floyd–Warshall algorithm originally applied to find a shortest path in a weighted graph. Afterward, we can determine the release date and due date of test j from $r_j = \Delta_{0j}$ and $d_j = p_j - \Delta_{j0}$, respectively.

We also have a condition for a partially ordered destructive test. For two different tests $j, k \in J$, we use the notation $j \simeq k$ to state that either test k precedes test j or both tests are executed on different prototypes. Also, $J_{\text{Last}} \subseteq J$ is the set of crash tests that must not be followed by any other test on the same prototype.

Due to the limited capacity in the workshop, we can produce k_b prototypes within the duration of p_b. Therefore, prototype $i \in I$ is not available before the time $a_i = p_b * \lfloor (i-1)/k_b \rfloor$.

We assume that the costs of the various prototype variants are roughly similar. Therefore, we want to minimize the number of required prototypes $m_r \leq m$. That means we may not need all prototypes of I. Only m_r prototypes must be constructed for processing all tests. Solving this optimization problem is equivalent to searching for the minimum number of prototypes which leads to a valid feasible schedule that observes all constraints.

Based on the standard scheduling framework, this problem is an extension of scheduling identical machines in parallel. We can use the notation $P_m|r_j, d_j, M_j, prec|m_r$ to represent our problem. However, this notation does not include all facets of the problem like the selection of a variant for each prototype, as well as the application-specific conditions.

Notice also that when we further neglect the machine eligibility and temporal constraints, our problem can be considered a generalization of the bin packing problem which minimizes a number of bins (or machines) to store items (or jobs) with various sizes (or processing times).

4 Constraint programming approach

Constraint programming (CP) is a programming platform describing relations between variables in the form of constraints. CP was originally developed in computer science and successfully applied to solve various kinds of scheduling problems [1]. CP consists of two main parts: a modeling part that generates a set of constraints to be satisfied and a search part that describes how to search for solutions.

Furthermore, CP applies a mechanism called constraint propagation to reduce the domain of variables. However, this reduction is usually not enough to obtain a fea-

sible solution. The solver has to search for an assignment of values to variables such that all constraints are satisfied. When propagation algorithms find an inconsistency during the search, CP has a functionality called backtracking to add non-redundant constraints and keep trying other possible decisions. The reduction and the search parts must simultaneously interact in order to achieve a feasible solution. We call this process of solving a constraint satisfaction problem (CSP).

CP has been extended to find not only any feasible solution, but also an optimal solution to a constraint optimization problem (COP). The solving mechanism is based on the branch-and-bound principle. As a bound on the objective is added, CP will find a new solution whose objective value is strictly better than the current one. As a result, the objective of 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 logic expressions, like *if…else*. Thus, we need not use many complex linear inequalities to represent the machine eligibility constraints. This leads to a compact formulation of our complex scheduling problem. In addition, we can enhance the performance of a CP solver by providing a problem specific search strategy in which the experience from using heuristics can be applied.

In this section, we apply CP to formulate all requirements of our scheduling problem with the objective function to minimize the number of required prototypes. Also, we develop several branching schemes in order to improve the performance of the search algorithm.

4.1 CP formulation

To formulate our CP model, we first define the following variables:

- $v_i \in V \; \forall i \in I$ represents the variant of prototype i
- $x_j \in I \; \forall j \in J$ indicates the allocation of test j to one of the prototypes
- t_j is the start time of test j

In this problem, we consider prototype i as an alternative resource. Each test can be allocated to one of the suitable prototypes in the set I known as the set of alternative resources. Also, each prototype can perform at most one test at the same time. Thus, we must maintain disjunctive relations between tests allocated to the same prototype.

In general, Baptiste et al. [1] viewed the alternative resource constraints as being propagated as if job j were split into m fictive jobs where each fictive job requires machine i. Constraint propagation will deduce new bounds like earliest or latest start times. When the bounds of fictive job j on machine i become incoherent, machine i must be removed from the set of alternative resources for job j.

We present our CP formulation as follows:

$$\text{Minimize} \quad \max_{j \in J} x_j \tag{1}$$

subject to:

$$v_i \notin M_j \Rightarrow x_j \neq i \quad \forall i \in I, \; \forall j \in J \tag{2}$$

$$t_j \geq r_j \quad \forall j \in J \tag{3}$$

$$t_j + p_j \leq d_j \quad \forall j \in J \tag{4}$$

$$(x_j = x_k = i) \Rightarrow (t_j + p_j \leq t_k) \vee (t_k + p_k \leq t_j)\ \forall i \in I, \forall j, k \in J,\ j \neq k \tag{5}$$

$$x_j = i \Rightarrow t_j \geq a_i \quad \forall i \in I,\ \forall j \in J \tag{6}$$

$$t_j - t_k \geq \delta_{kj}\ \forall j, k \in J \cup \{0, n+1\},\ j \neq k \tag{7}$$

$$x_j = x_k \Rightarrow t_j \geq t_k + p_k\ \forall j \simeq k,\ \ j, k \in J \tag{8}$$

$$x_j = x_k \Rightarrow\ t_j \geq t_k + p_k\ \forall j \in J_{\text{Last}}, \forall k \in J \setminus J_{\text{Last}}. \tag{9}$$

The objective function (1) minimizes the maximum number of prototypes used to allocate all tests. Constraint (2) prevents a test from being assigned to a prototype whose variant does not belong to the eligibility set of the test. Constraint (3) and Constraint (4) state that tests must be performed between their release and due dates. Constraint (5) is the alternative resource constraint to ensure either test j must be completed before starting test k or vice versa if both tests are allocated to the same prototype i.

Constraint (6) ensures that a test on machine i cannot start before the availability of the prototype. Constraint (7) considers the time lag constraints. Constraint (8) and Constraint (9) are provided for the partially ordered destructive tests and the crash tests, respectively.

Note that, our CP solver named ILOG Scheduler 6.2 [9] provides several filtering algorithms for the alternative resource constraints. We previously found that it is important to select appropriate constraint propagation rules [14]. First, we used the lowest propagation strength containing only a light precedence graph and disjunctive constraints. However, both propagation rules could not find any solution even for a small instance with around 40 tests. Therefore, an edge-finding technique had to be applied. Also, we tried to use the strongest propagation level containing a balance constraint. However, the computation cost was so high that our computer with 4 GB RAM ran out of memory when solving large instances with hundreds of tests.

4.2 Search strategy

Although some search algorithms are already provided inside CP solvers, these rules are generally designed to handle various kinds of problems. Particularly when a problem contains several hundreds of variables, it becomes very important to implement an efficient search algorithm to specifically solve the problem.

In this paper, we apply a chronological backtracking search to try different values in the domains of variables. As a search tree starts from a root node, we specify which variables to branch on further. The selected variable must be then instantiated to one of the values in its current domain. This is known as a labeling process to specify a variable-value pair. The whole search tree can be determined by the ordering of variables to be explored and the ordering of values to be assigned.

At each node, the assignment to an uninstantiated variable must satisfy the constraints which are involved. When the constraint check fails, another domain value of the variable must be tried. If there is no more possible values in the domain to be tried, the algorithm has to backtrack from a dead end to the most recently

instantiated variable. A solution is achieved when the last variable can be instantiated while all constraints are satisfied. A depth-first search is applied to explore the search tree since we aim to obtain a good feasible solution as soon as possible.

Note that we can apply CP with other branching schemes and search strategies as introduced by van Beek [25]. For instance, domain splitting does not try to instantiate a variable, but repeatedly splits the domain of the current variable into subdomains containing no duplicate values. Also, it is not necessary to retract the most recently instantiated variable. Backjumping aims to discover and retract the branching possibly leading to a dead end. Unlike the depth-first search, a discrepancy-based search is proposed to help recover the mistake made early in the search.

The main principle of our branching schemes is to allocate critical tests first. Obviously, it is difficult to schedule tests which require specific prototype variants and have tight temporal restrictions. We should reserve prototypes for these critical tests before considering other tests.

In fact, our technique is extended from the least flexible job first (LFJ) rule commonly applied to solve scheduling problems with a machine eligibility constraint [22]. The LFJ rule first selects a job (or a test) having the smallest number of eligible machines (or prototypes). In addition, we take into account the temporal constraints.

At the beginning, we focus on how to allocate tests to suitable prototypes. That means we have to select variables x_j and assign their values. Our test selection method is initially based on a minimum domain strategy over variables x_j. After that, we consider domains of start time variables and try to select tests using the minimum latest start time or the minimum slack time.

In this section, we propose several criteria for tests and prototype selection methods. Their combinations result in different kinds of test allocation process. Afterward, we present a complete branching scheme which includes a common procedure to branch on the remaining variables, i.e. variants of prototypes v_i and start times of tests t_j.

4.2.1 Test selection method

To select a critical test, we consider the number of remaining possible prototypes as the first criterion. During the search, the domain of x_j shrinks due to constraint propagation. 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. When there is more than one test having the minimum number of remaining possible prototypes, we suggest five alternative rules based on temporal conditions for breaking ties.

- Slack
  Tests are selected using the minimum slack time ($lst_j - est_j$), where est_j and lst_j represent the earliest and latest start times. Their domains are dynamically changed due to constraint propagation during the search. Clearly, a test with a small slack time tends to be more difficult to schedule.
- LST
  Tests are selected using the minimum latest start time. Due to the construction time of prototypes, the number of prototypes available at the beginning are limited. Thus, we give more priority to tests with earlier latest start times.

- Slack-LST
  As formally described in Algorithm 1, we first determine the set J_m containing unscheduled tests whose numbers of remaining possible prototypes are minimal. Given the set J_m, we select tests satisfying the condition $lst_j - est_j < c' p_b$ and add them to the set J_l. The minimum slack time rule is applied to select a test in the set J_m if the set J_l is still empty. Otherwise, we select a test in the set J_l using the minimum latest start time. By setting the parameter $c' = 1$, 2, and 3 in the algorithm, we obtain the rules: Slack-LST1, Slack-LST2, and Slack-LST3, respectively.
  Instead of relying on the normal slack rule, this algorithm immediately schedules a test with the minimum latest start time when the slack times of some tests become less than $c' p_b$ or our time gap allowing for the construction of new prototypes.
- LST-slack
  After the set J_m is initialized, we select tests satisfying the condition $lst_j - est_j < c' p_j$ and add them to the set J_s. The minimum latest start time rule is applied to select a test in the set J_m if the set J_s is still empty. Otherwise, we select a test in the set J_s using the minimum slack time. The procedure is formally described in Algorithm 2. By setting the parameter $c' = 1, 2$, and 3 in the algorithm, we obtain the rules: LST-Slack1, LST-Slack2, and LST-Slack3, respectively.
  Instead of relying on the normal LST rule, we prefer a test with the minimum slack time when there are some tests whose slack times become relatively less than their processing times.
- Backward planning
  We further implement a search scheme using problem-specific knowledge. As suggested by Bartels and Zimmermann [3], it is beneficial to apply a backward planning scheme especially in this case which consists of many crash tests and partially destructive tests.
  In this case, we want to find tests which can be executed as late as possible. The maximum latest start time is considered the second criterion after the number of possible machines. Also, crash tests and partially ordered destructive tests have higher priority than the other tests. The backward planning rule is formally described in Algorithm 3.

Algorithm 1 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 any test k having the minimum number of possible prototypes, and add it to J_m
From J_m, find any test k whose $lst_k - est_k < c' p_b$, and add it to J_l
if $J_l \neq \emptyset$ **then**
  From J_l, arbitrarily select any test j with the minimum latest start time
else
  From J_m, arbitrarily select any test j with the minimum slack time
end if
Return test j

Algorithm 2 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 any test k having the minimum number of possible prototypes, and add it to J_m
From J_m, find any test k whose $lst_k - est_k < c' p_k$, and add it to J_s
if $J_s \neq \emptyset$ **then**
  From J_s, arbitrarily select any test j with the minimum slack time
else
  From J_m, arbitrarily select any test j with the minimum latest start time
end if
Return test j

Algorithm 3 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 any test k having the minimum number of possible prototypes, and add it to J_m
From J_m, find any test k with the maximum latest start time, and add it to J_t
if $J_t \cap J_{\text{Last}} \neq \emptyset$ **then**
  Arbitrarily select any 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 any partially ordered destructive test j in J_t
else
  Arbitrarily select any test j in J_t
end if
Return test j

4.2.2 Prototype selection method

After a critical test is found, it must be allocated to an appropriate prototype. We suggest 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 occupied by other tests 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 due to production capacity.
  The MaxSt rule is formally described in Algorithm 4. The set J_b consists of tests which are already allocated to one of the prototypes in the 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 also means

$I_p \cap I_r \neq \emptyset$, where I_p is the set of all prototypes which can perform test j. We then select a prototype by considering the maximum latest start time.
We have to further mention 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 prototypes not being used before. As a result, a prototype with the minimum index must be selected.

Algorithm 4 Prototype selection method: MaxSt

%Given J_a be the set of tests not yet allocated and the selected test j
%Determine the set of allocated tests $J_b = J \setminus J_a$
%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**
  Arbitrarily select any prototype i which can perform test j at the maximum latest start time
else
  Select new prototype $i \in I_p$ with the minimum index
end if
Return prototype i

4.2.3 Complete branching schemes

First for the test allocation process, we combine the proposed test and prototype selection methods in order to achieve various branching schemes. For instance, Algorithm 5 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_a) finds prototype i to perform test j by applying the MaxSt selection method.

Second, 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 domains of variables v_i start with a complete range $[1, ..., l]$ that is, prototypes can be assigned to any variant. After the test allocation process, the domains of variables v_i automatically shrink as the constraint propagation removes variants which cannot satisfy the component requirements of the assigned tests.

Finally, the start time of each test is determined based on the schedule-or-postpone method [12]. The algorithm selects an unscheduled job with minimal earliest start time and tries to schedule it as early as possible. However during backtracking, the job is postponed. This means that the algorithm can neglect this job until its domain of earliest start time is increased due to the constraint propagation mechanism.

To implement the last part in ILOG Scheduler 6.2, we can apply the function IloSetTimesForward. In general, this command can lead to an incomplete search when there exists a precedence constraint with negative delay, or when the processing time of a job depends on its start or end time, or when reservoir resources are used [9]. These particular situations do not happen in our scheduling problem.

5 Hybrid approach

Our test scheduling problem can be decomposed into master and slave 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 legally perform all tests. This problem structure suggests a hybrid scheme based on a form of Benders decomposition in which the master and slave problems are solved by MILP and CP, respectively, since MILP is a suitable tool for optimization, while CP has the capability to quickly find a feasible solution. Thus, MILP and CP methods can appropriately complement each other.

Algorithm 5 Slack/MaxSt branching scheme

```
%PART 1) Assign Tests to Prototypes
%Let J_a be the set of tests not yet allocated
J_a := J
while J_a ≠ ∅ do
   j = SelectTest(Slack, J_a)
   i = SelectPrototype(MaxSt, j, J_a)
   Try
    x_j = i
   or
    x_j ≠ i (for backtracking)
   J_a = J \ {j}
end while
%PART 2) Assign Prototype Variants
while not all prototypes have specified variants do
   – Arbitrarily select any prototype i
   – Instantiate v_i with any possible value in its remaining domain
end while

%PART 3) Assign Start Times of Tests
%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 ≠ ∅ then
      – Arbitrarily select any test j from J_s
      – 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
```

An optimal solution is obtained when CP achieves a complete schedule corresponding to a solution to the simplified planning problem previously found by MILP. However, when CP fails to find any feasible solution, the planning problem must be reformulated such that invalid solutions to previous planning problems are excluded. To remove those solutions, Benders cut constraints are generated and augmented to the master problem. We can prove the whole problem infeasible if Benders cuts eliminate every feasible solution to the master problem.

In this section, we apply a procedure suggested by Bartels [2] to identify pairs of tests which must definitely be executed on different prototypes. As a result, we can formulate a maximum clique problem to determine a lower bound of the number of required prototypes. Afterward, the planning MILP model is developed based on set covering and energetic reasoning principles in order to estimate the variant and demand of prototypes occurring in relevant time intervals.

Also, we formulate various CP models for the scheduling process. One of them is constrained to obey the decision from the planning level, while the others are not so restrictive and able to change the given decision as long as the demand of prototypes remains the same. Since CP is normally efficient in finding a feasible solution, it could help us to finish the solving process sooner.

We further describe the limitations of the hybrid approach which occur when solving very large instances. The size of the planning MILP model can exceed the memory capacity, while the search space for CP becomes so large and cannot be completed within a reasonable time. We present two additional procedures to alleviate these obstacles.

5.1 Procedure for finding an initial lower bound

First, we define the term $j||k$ for two different tests j and k which certainly cannot be executed on the same prototype. To determine these overlapping tests, we apply the following criteria:

- Test j and test k always use different prototype variants, i.e. $M_j \cap M_k = \emptyset$.
- Test j and test k must overlap each other due to temporal conditions, i.e. $\Delta_{jk} > -p_k$ and $\Delta_{kj} > -p_j$, as represented in Fig. 1.
- Test k cannot finish before a crash test j, i.e. $j \in J_{\text{Last}}$ and $\Delta_{jk} > -p_k$.
- Test j and test k are both crash tests, i.e. $j, k \in J_{\text{Last}}$.
- Test j and k are partially destructive to each other, i.e. $j \simeq k$ and $k \simeq j$.
- Test j is partially destructive to test k and test k cannot finish before test j, i.e $j \simeq k$ and $\Delta_{jk} > -p_k$.
- Test j is partially destructive to a crash test k, i.e. $j \simeq k$ and $k \in J_{\text{Last}}$.

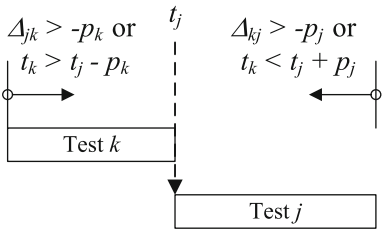

Fig. 1 Domains of start times of overlapping tests

Following Pardalos and Xue [21], the maximum clique problem is further defined. Let $G_c = (J, E)$ be an undirected graph, where an edge set is given by $E = \{(j,k) | j,k \in J \text{ and } j||k\}$. The complement of $G_c = (J, E)$ is the graph $\bar{G}_c = (J, \bar{E})$, where $\bar{E} = \{(j,k) | j,k \in J,\ j \neq k \text{ and } (j,k) \notin E\}$. A clique is a subset of J, where every pair of its vertices is connected by the given edge E. The maximum clique problem is to determine a clique having the largest possible size.

We apply integer programming to solve the maximum clique problem. Let a decision variable u_j become one if node j belongs to a clique of G_c.

$$\text{Maximize} \quad \sum_{j \in J} u_j \tag{10}$$

subject to

$$u_j + u_k \leq 1 \quad \forall (j,k) \in \bar{E} \tag{11}$$

Also, we define $U = \{j \in J | u_j = 1\}$ to represent the set of the obtained solution. Therefore, the initial lower bound of m_r must be greater than or equal to $|U|$.

5.2 Planning MILP model

First, we notice that our problem can be considered a set covering problem if the planning problem determines the minimum number of prototypes subject to only the component requirements of all tests. However, it is important to include other characteristics of the slave problem in the master MILP model in order to avoid trivial infeasible solutions.

Previously, Hooker [6] suggested the relaxation of the MILP model in order to minimize the number of late jobs according to the specified capacity of the cumulative resources. In contrast, our problem aims to minimize the peak of resource utilization such that no test is late.

In general, this concept is similar to the energetic reasoning principle applied to propagate constraints in cumulative resources [1]. We successfully encode this principle into our planning MILP model. In addition, our model tries to approximate the additional demand of prototypes as some of prototypes cannot be used before their available times and after processing crash tests.

First, we define decision variables for the planning MILP model. Notice that we use an array of binary variables to represent the number of prototypes for each variant instead of using one integer variable in order to comply with the logic-based Benders cut. For each variant, g represents the maximum number of prototypes that can be built. Also let h denote an index for set $H = \{1, ..., g\}$. The variables are provided as follows:

- $w_{v,h} = 1$, iff h prototypes are required for variant v.
- $y_{v,j} = 1$, iff test j is assigned to a prototype with variant v.
- $z_{v,i} = 1$, iff prototype i is assigned to variant v.

Let us define set $R_v = \{r_j \mid j \in N_v\}$ containing the distinct elements of release dates of tests which can be performed by variant v. Similarly, set $D_v = \{d_j \mid j \in N_v\}$ contains only the distinct elements of due dates of tests which can be performed by

variant v. Using parameters $t_1 \in R_v$ and $t_2 \in D_v$ with $t_1 < t_2$, we define the following sets:

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

We represent the concept of energetic reasoning in Fig. 2. For each variant and every time interval, we must provide at least enough resources corresponding to the expected job consumptions. The necessary condition is provided as follows:

$$\sum_{h \in H} hw_{v,h}(t_2 - t_1) \geq \sum_{j \in J_c(t_1,t_2)} y_{v,j} p_j + \sum_{\substack{j \in J_l(t_1,t_2) \\ \cup J_{ol}(t_1,t_2)}} y_{v,j}(r_j + p_j - t_1) + \sum_{\substack{j \in J_r(t_1,t_2) \\ \cup J_{or}(t_1,t_2)}} y_{v,j}(t_2 - d_j + p_j)$$
$$+ \sum_{j \in J_{oa}(t_1,t_2)} y_{v,j}(t_2 - t_1) + \sum_{j \in J_{\text{Last}} | d_j < t_1} y_{v,j}(t_2 - t_1) + \sum_{i \in I | t_1 < a_i < t_2} z_{v,i}(a_i - t_1)$$
$$+ \sum_{i \in I | a_i \geq t_2} z_{v,i}(t_2 - t_1) \quad \forall v \in V, \forall j \in N_v, \forall t_1 \in R_v, \forall t_2 \in D_v \qquad (12)$$

We can realize that variables $y_{v,j}$ decide which jobs must be considered with respect to a prototype variant. The term with $J_c(t_1, t_2)$ specifies the sum of processing times of tests which start and finish in this interval. Afterward, $J_l(t_1, t_2)$ considers some parts of tests which can be left-shifted to start at their earliest start times r_j but still cannot finish before t_1. Hence, the overlapping part of at least $r_j + p_j - t_1$ can be included. Similarly, tests in $J_r(t_1, t_2)$ require at least $t_2 - d_j + p_j$. Also, we apply either left-shifting or right-shifting to other tests in order to determine the

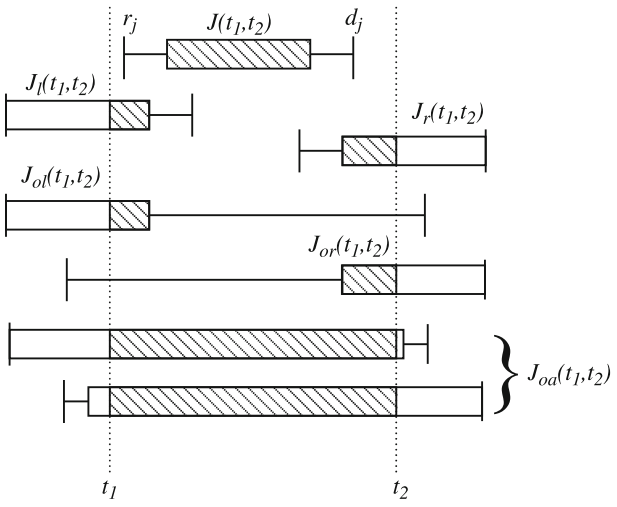

Fig. 2 Representation of the energetic reasoning concept

minimum contributing value ($\max\{0, \min\{r_j + p_j - t_1, t_2 - d_j + p_j, t_2 - t_1\}\}$). In this category, tests in J_{ol} and J_{or} need their minimum energy requirement when they are left- and right-shifted, while J_{oa} consumes the whole period between t_1 and t_2.

Additionally, we consider the impact of each crash test in J_{Last} as if it still occupies a prototype after its due date. Also, the prototype construction time a_i can either partially overlap into the considered period or completely consume the entire gap if the prototype is not yet ready.

Our previous study compared the performance of the hybrid approach using two MILP models [14]. The first model considered only tests in J_c, while the second model included all tests in the other remaining sets. Our results show that the second model used a lower number of iterations required to augment Benders cuts and can decrease overall computational time.

Note that this study was based on few real-life instances containing up to almost 500 tests and containing different additional conditions. When we deal with larger problems given in this paper, we also need a preliminary procedure as discussed later in Section 5.5.

In the following, we present our MILP formulation together with the necessary condition from energetic reasoning.

$$\text{Minimize} \quad \sum_{v \in V} \sum_{h \in H} hw_{v,h} \tag{13}$$

subject to:

Constraint (12)

$$\sum_{h \in H} w_{v,h} \leq 1 \quad \forall v \in V \tag{14}$$

$$\sum_{v \in M_j} y_{v,j} = 1 \quad \forall j \in J \tag{15}$$

$$\sum_{h \in H} w_{v,h} \geq y_{v,j} \quad \forall j \in J, \forall v \in V \tag{16}$$

$$\sum_{v \in V} z_{v,i} \leq 1 \quad \forall i \in I \tag{17}$$

$$\sum_{i \in I} z_{v,i} = \sum_{h \in H} hw_{v,h} \quad \forall v \in V \tag{18}$$

$$\sum_{v \in V} \sum_{h \in H} hw_{v,h} \geq |U| \tag{19}$$

$$\sum_{h \in H} hw_{v,h} \geq \sum_{j \in N_v \cap U} y_{v,j} \quad \forall v \in V \tag{20}$$

$$\sum_{h \in H} hw_{v,h} \geq y_{v,j} + y_{v,k} \quad \forall j || k, \forall v \in V \tag{21}$$

The objective function (13) is to minimize the total number of prototypes. As the number of prototypes required for each variant is represented by an array of binary

variables, Constraint (14) ensures that at most one variable in the array H can be equal to one. Constraint (15) assigns each test to exactly one variant belonging to the eligible set of the test. Consequently, Constraint (16) guarantees that at least one prototype is built if the variant is selected for any test.

Moreover, Constraint (17) ensures that each prototype belongs to at most one variant. Note that some prototypes may remain unassigned to any variant, i.e. $\sum_{v \in V} z_{v,i} = 0$, since we may not need all prototypes of I. Constraint (18) indicates that the number of prototypes for each variant found by variables $z_{v,i}$ and $w_{v,h}$ are equal.

Constraint (19) specifies a lower bound using the solution to the maximum clique problem. As tests in the clique cannot be executed in parallel, Constraint (20) determines the minimum number of prototypes for each variant. Similarly, Constraint (21) demands at least two vehicles being available for tests which must be done on different prototypes.

We further mention that Benders decomposition is commonly applied to separate between the master problem assigning jobs (or tests) to machines (or prototypes) and the slave problem scheduling jobs in each machine. However, our problem is quite extraordinary since the master problem has to deal with the allocation of jobs to suitable types of machines (or prototype variants). Also, the number of machines is not just fixed but has to be minimized such that a valid schedule can be finally obtained.

In Section 6.2, our computation results show that we have to solve quite large master MILP and large slave CP models despite the decomposition. Assume we additionally consider the allocation of tests to prototypes in the master problem in order to reduce the complexity of slave problems. Certainly, the MILP model will require even more variables, constraints and computation capacity.

Furthermore, Baptiste et al. [1] discuss other filtering algorithms for the propagation of resource constraints. We tried to include the Time-Table constraint into the planning MILP model but could not improve the solving performance. We cannot apply other algorithms like Edge-Finding and Not-First/Not-Last since they need a large amount of memory for variables and constraints to identify sequential relations between tests.

5.3 Scheduling CP models and benders cuts

In this section, we introduce three scheduling CP models and their corresponding Benders cuts. Clearly, these models are similar to the formulation presented in Section 4 for solving the complete problem. The objective function to minimize the peak demand can be neglected since it is sufficient to find a feasible solution.

For the first scheduling CP model, additional constraints are required such that CP only extends the solution obtained from the planning MILP model. We further suggest two alternative models allowed to partially change the decision from MILP. In total, we present three kinds of hybrid approaches corresponding to various scheduling CP models.

5.3.1 Hybrid approach with fixed allocation of test (HF)

There are two aspects covered by the solutions to the planning MILP model: $y_{v,j}$ suggests the allocation of tests to variants; and $z_{v,i}$ indicates which variants are built

according to the production sequence. The scheduling process is required to further determine the allocation of tests to prototypes x_j and the start time t_j. The scheduling CP model for HF is given as follows.

Solve:

Constraint (3) – (9)

$$y_{v_i,j} \neq 1 \Rightarrow x_j \neq i \quad \forall i \in I, \forall j \in J \tag{22}$$

$$z_{v,i} = 1 \Rightarrow v_i = v \quad \forall i \in I, \forall v \in V \tag{23}$$

We can realize that variables $y_{v,j}$ from the master problem restrict tests to be performed on prototypes with the specified variants instead of other possible prototype variants. Therefore, the component requirement Constraint (2) is replaced by Constraint (22). Moreover, Constraint (23) states that the sequence of prototype variants in the scheduling CP model must comply with the result obtained from the MILP model.

Next, we present the Benders cut constraint to eliminate invalid solutions from the previous iterations. The constraint will be incorporated into the master MILP problem if the solutions from previous steps are found infeasible by the scheduling CP model.

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

$$\sum_{v\in V}\sum_{h\in E^k_v} w_{v,h} + \sum_{v\in V}\sum_{j\in F^k_v} y_{v,j} + \sum_{v\in V}\sum_{i\in G^k_v} z_{v,i} \leq \sum_{v\in V}(|E^k_v| + |F^k_v| + |G^k_v|) - 1 \tag{24}$$

where $E^k_v = \{h \in H | w^k_{v,h} = 1\}$, $F^k_v = \{j \in J | y^k_{v,j} = 1\}$, and $G^k_v = \{i \in I | z^k_{v,j} = 1\}$. Constraint (24) considers only variables $w_{v,h}$, $y_{v,j}$, and $z_{v,i}$ whose values are equal to one at iteration k. We prohibit a new solution from simultaneously having the same set of these variables.

Note that we must specifically determine the test allocation to each prototype since the allocation of the crash tests and the partially ordered destructive tests has impact on the sequence of the following tests. Otherwise, we can basically consider a group of prototypes with the same variant as one cumulative resource with the capacity $C_v = \sum_{h\in H} hw_{v,h}$. Furthermore, we cannot separately solve a sub-problem per each variant as temporal constraints between a pair of tests can be allocated to prototypes with different variants.

5.3.2 Hybrid approach with changeable allocation of test (HT)

For the HT hybrid method, we allow the slave CP model to choose any suitable variant for each test rather than the variant specified by Constraint (22). The scheduling CP model for HT is given as follows.

Solve:

Constraint (2) – (9)

$$z_{v,i} = 1 \Rightarrow v_i = v \quad \forall i \in I, \forall v \in V \tag{25}$$

Instead of using the *nogood* constraint (24), we can generate a stronger Benders cut:

$$\sum_{v\in V}\sum_{h\in E_v^k} w_{v,h} + \sum_{v\in V}\sum_{i\in G_v^k} z_{v,i} \leq \sum_{v\in V}(|E_v^k| + |G_v^k|) - 1 \tag{26}$$

Constraint (26) considers only variables $w_{v,h}$ and $z_{v,i}$ whose values are equal to one at iteration k. The slave CP model has to consider all possible allocations between tests and variants. With this Benders constraint we can reduce a number of iterations to repeat solving the master problem. However, it comes with the cost of solving a more difficult CP model.

Notice that this slave CP model does not require the result of the variables $y_{v,j}$ obtained from the master problem. We can realize there is some redundant work coupling between the slave and master problems. Nevertheless, we cannot remove the variables $y_{v,j}$ from the MILP model since they are necessary for the formulation of several important constraints to determine the total number of required prototypes.

5.3.3 Hybrid approach with changeable sequence of prototype production (HP)

The result of the planning MILP model suggests which prototype variants are selected and in which order of production they must be built. However, when prototypes are only built earlier or later, this does not have any impact on the total number of required prototypes. We further allow CP to alternate the sequence of prototype production. The scheduling CP model for HP is as follows:

Solve:

Constraint (2) − (9)

distribute(**card**, **value**, **var**) (27)

The first array **card** $= [w_1, .., w_l]$ represents the number of prototype variants determined from the result of the planning MILP model, i.e. $w_v = \sum_{h\in H} hw_{v,h}$ for each prototype variant $v \in V$ where $|V| = l$. Also, the second array **value** $= [1, .., l]$ represents the list of prototype variants $v \in V$. The last array **var** $= [v_1, ..., v_m]$ contains the CP variables $v_i \in V\ \forall i \in I$ where $|I| = m$.

The notation **distribute** is a global cardinality constraint in a CP platform. The number of prototype variants specified by $w_v\ \forall v \in V$ must be equal to the number of occurrences of the value v in the array **var**. Therefore, Constraint (27) ensures that the number of prototypes of variant v found in the slave CP is equal to the estimated demand given by the master MILP.

Finally, we introduce another Benders cut constraint:

$$\sum_{v\in V}\sum_{h\in E_v^k} w_{v,h} \leq \sum_{v\in V}|E_v^k| - 1 \tag{28}$$

Constraint (28) considers only variables $w_{v,h}$ whose values are equal to one at iteration k. The slave CP model has to consider all possibilities of the allocation between tests and variants as well as the production sequence of prototypes. Therefore, we can prohibit a new solution having the same number of prototypes required for each variant as given by the MILP variables $w_{v,h}$.

5.4 Additional benders cuts

To further improve the performance, we aim to eliminate other possible failures that might occur. We formulate sub-CP models for each prototype variant instead of solving just the scheduling CP model. Each sub-CP model considers only a cumulative resource constraint with its corresponding resource capacity and the set of allocated tests. We also neglect temporal constraints between two tests if both of them are allocated to different prototype variants. Thus, we can check the feasibility of each sub-CP model separately.

The purpose of these cuts is to quickly check cumulative constraints in small sub-CP models. We can effectively prevent many invalid solutions due to the resource constraints. The scheduling CP model is necessary to finally verify all conditions together as shown in Fig. 3.

Assume the sub-CP model of variant v' is found infeasible, that means the number of available prototypes and the production sequence of prototypes are not appropriate to execute all the tests in the given set. As the processing times are always constant, this result can be applied to any other variant. This leads to another Benders cut:

$$\sum_{h \in E^k_{v'}} w_{v,h} + \sum_{j \in F^k_{v'}} y_{v,j} + \sum_{i \in G^k_{v'}} z_{v,i} \leq |E^k_{v'}| + |F^k_{v'}| + |G^k_{v'}| - 1 \quad \forall v \in V. \tag{29}$$

where $E^k_{v'} = \{h \in H | w^k_{v',h} = 1\}$, $F^k_{v'} = \{j \in J | y^k_{v',j} = 1\}$, and $G^k_{v'} = \{i \in I | z^k_{v',j} = 1\}$ indicate the number of prototypes with variant v', the allocation of tests to variant v', and the production sequence of prototypes with variant v', all of which lead to infeasibility. Constraint (29) must eliminate the same set of these variables to recur on other prototype variants.

Although it takes additional effort to solve these small sub-CP models, we can reduce the number of iterations and the computation time for solving the large master MILP model. In fact, we can solve the sub-CP models before considering the whole scheduling CP model. The complete schedule cannot happen as well if at least one of sub-CP models is not feasible.

Moreover, the available time of a prototype can be considered a dummy test competing for the resource like other normal tests. Assume a sub-CP model is found infeasible for tests allocated to the second prototype, i.e. $i = 2$ in the production sequence. This set of test allocations will certainly fail again if allocated to the third prototype due to the increase of its available time. Therefore, we must avoid a similar failure to occur on any succeeding prototypes via a constraint:

$$\sum_{i \in I} a_i z_{v,i} < \sum_{i \in G^k_{v'}} a_i z_{v,i} + M(|E^k_{v'}| + |F^k_{v'}| - \sum_{h \in E^k_{v'}} w_{v,h} + \sum_{j \in F^k_{v'}} y_{v,j}) \quad \forall v \in V \tag{30}$$

where M is a sufficiently large positive number, e.g. $M = \sum_{i \in I} a_i$. Constraint (30) becomes redundant when variables $w_{v,h}$ in the set $E^k_{v'}$ and variables $y_{v,j}$ in the set $F^k_{v'}$ do not all become one. Otherwise, the production sequence of prototypes determined from the set $G^k_{v'}$ must be modified such that the sum of the available times of prototypes decreases.

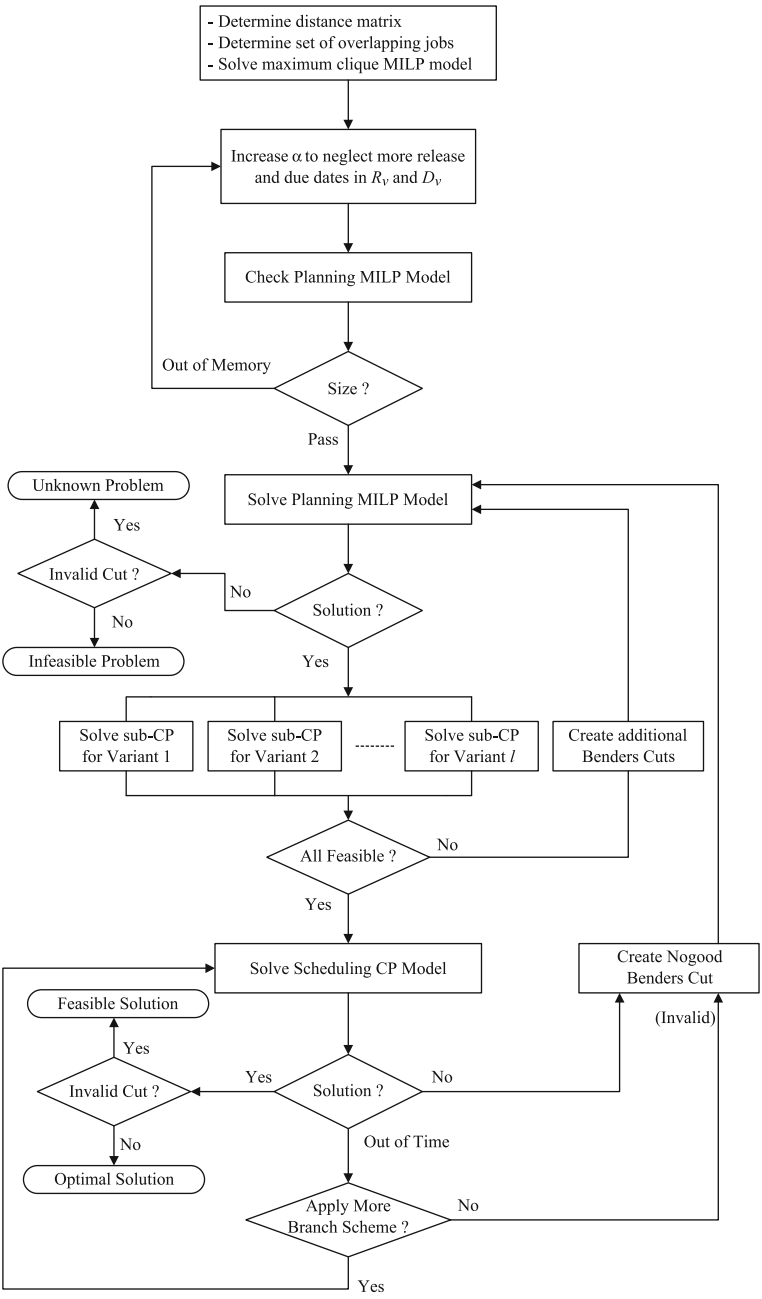

Fig. 3 Full diagram of MILP-CP hybrid approach

5.5 Procedure for decreasing the size of the planning MILP model

Although it is quite important to have a planning model as accurate as possible, the model size can substantially grow beyond memory capacity. Based on energetic

reasoning, the size of our model depends on the number of release dates and due dates in R_v and D_v. These numbers can be very large particularly in case of randomly generated instances. Notice that when we previously solved the real-life instances, we need not apply this procedure since the release dates and due dates tend to be assigned in a duplicate fashion to specific milestones [14].

For this case, we have to select only some of the values leading to the solvable MILP model. As shown in Algorithm 6, each release date is sequentially considered by using the parameter α to specify the size of its neighborhood. We include r_j only when no values within its neighborhood belong to R_v. The same procedure is applied to consider each d_j and to determine D_v.

As α starts from zero, it means we try to create a MILP model with all possible values of release dates and due dates. When the obtained MILP model becomes too large, we iteratively increase α by one to reduce the sizes of R_v and D_v. The procedure repeats until the MILP model becomes solvable.

Algorithm 6 Procedure for decreasing the size of the planning MILP model

```
Set α = 0, R_v = ∅ and D_v = ∅
repeat
  for v = 1 to l do
    for j = 1 to n do
      Set R_e = ∅ and D_e = ∅
      for a = −α to α do
        R_e = R_e ∪ {r_j + a}
        D_e = D_e ∪ {d_j + a}
      end for
      if R_e ∩ R_v = ∅ and v ∈ M_j then
        R_v = R_v ∪ {r_j}
      end if
      if D_e ∩ D_v = ∅ and v ∈ M_j then
        D_v = D_v ∪ {d_j}
      end if
    end for
  end for
  α = α + 1
until MILP model using values in R_v and D_v can be solved
```

5.6 Procedure for increasing search regions of the scheduling CP model

Although the complexity of the scheduling CP model reduces after the Benders decomposition, the problem may be still too large especially for proving feasibility within an appropriate limited time.

As CP cannot explore all nodes in the search tree, we show later in the computation section that it is more efficient to apply various branching schemes. We should not waste time relying on only a single branching scheme. Due to the depth-first search strategy, CP can get stuck in the bottom part of the search tree. From our experience solving the complete problem using CP, it is better to stop searching after a reasonable time limit and restart with another branching scheme.

To allow CP to explore several potential regions, we consecutively solve the same scheduling CP model using a series of search schemes. After all search schemes cannot find a complete solution, we stop solving the CP problem and include a Benders cut to the MILP model. Since the search process is incompletely terminated, the Benders cut can be considered invalid since it may eliminate an optimal solution. Especially when a value of the objective function changes afterward, we can achieve only a feasible solution.

6 Computational results

We apply our CP and hybrid approaches to solve the benchmark given by Bartels and Zimmermann [3]. The standard tool named ProGen/max [24] was applied to randomly generate these instances based on the characteristics of a real scenario given from a manufacturer. There are two sets of data instances. The first set contains 100 instances, each of which has only 20 tests. To evaluate a real-size problem with 600 tests, we also use the second set containing 60 instances. For each instance, 10% and 30% of all tests are crash tests and partially ordered destructive tests.

Our computations are performed on a Pentium Dual-Core, 3.0 GHz, and 4 GB RAM. Our MILP and CP solvers are CPLEX 10.0 [8] and Scheduler 6.2 [9].

6.1 Results of solving small instances

First, CP is applied to solve the small instances and can obtain optimal solutions for all cases. The average solving time of 8.52 s is achieved by using the Slack/MinId search scheme. When we deploy other search schemes, the computation time is slightly different.

Afterward, we apply our hybrid approaches: HF, HT, and HP. Within the time limit of total computation at 1 h, only HP can optimally solve all instances. That means it becomes more robust to let CP immediately change parts of the decision suggested by the MILP model. The HP hybrid approach requires around 15.27 s on average.

We present the overall results in Table 1. The information of solving by MILP approach is provided by Bartels [2] and Bartels and Zimmermann [3]. Their heuristic and GA methods achieved the best solutions of about 10% and 2% far from the

Table 1 Computational results of solving 100 small instances by various approaches

Approach	# of instances solved within time τ				Computation time (s)	
	$\tau \leq 1$ s	$\tau \leq 10$ s	$\tau \leq 100$ s	$\tau \leq 1$ h	Average	Max
MILP	54[b]	93[b]	100[b]	100[b]	3.92[c]	73.81[c]
CP	67	89	98	100	8.52	323.14
HF	57	68	76	81	99.11[a]	3,411.72[a]
HT	69	83	90	97	91.11[a]	3,536.79[a]
HP	81	90	98	100	15.27	1,032.84

[a]From results obtained within 1 h

[b]From Bartels and Zimmermann [3]

[c]From Bartels [2]

Table 2 Speed-up ratio of solving 100 small instances

Method	# of instances with speed-up ratio between					
	0.001–0.01	0.01–0.10	0.1–1.0	1.0–10.0	10–100	100–1,000
HF	6	8	22	24	16	5
HT	2	14	17	46	14	4
HP	–	–	22	62	11	5

optimality. Their MILP solver is CPLEX 10.0 running on Pentium IV with 2.4 GHz. Certainly, it is not appropriate to compare the performance when using different computation platforms. However, the results suggest that the MILP approach is relatively faster on average. Nevertheless, within the computation time of 1 s our CP and hybrid approaches can solve 67 and 81 instances, compared with 54 instances done by MILP. We can realize that the remaining 30% of instances significantly reduce the performance of our CP and hybrid approaches.

We further analyze the performance when using CP and hybrid approaches. For each instance, we determine a speed-up ratio such as $T_{\mathrm{CP}}/T_{\mathrm{HP}}$, where T_{CP} and T_{HP} are the computation times of CP and HP hybrid methods. Table 2 presents the number of instances whose speed-up ratios lie in various ranges. There are almost 80% of instances which HP can solve faster than CP since their ratios are greater than 1.

However, the average computation time of HP is about twice as long as CP. We can explain using a simple example. Let us assume that we consider 5 instances, each of which CP can solve within 1 s. HP needs 0.01 s for four instances and 10 s for the last one, which results in the speed-up ratios of 100 and 0.1, respectively. Meanwhile, the average time of HP is roughly about 2 s.

Furthermore, Table 3 presents the number of iterations required by various hybrid approaches. HP needs about 3 iterations on average, while almost 60 instances are solved at the first iteration. Even for these small problems, HF and HT take a considerable number of iterations to correct the mistakes as the planning model cannot completely consider the impact of several constraints on the scheduling part.

It becomes more severe when not only the test allocation but also the estimation of prototype demand is wrong. We notice there are 32 instances in which MILP initially tries to suggest using one prototype below the optimal solutions. To increase the demand and get the correct solutions, HP requires about 7 iterations by average, while the number dramatically climbs up to 3,721 iterations for HF.

Until now, our results show that HP is more efficient when compared with HF and HT. However, there are still 10 and 25 instances for which HF and HT need less computation time. In this case, HF and HT require only 1.0 and 1.7 iterations on

Table 3 Number of iterations required for solving 100 small instances by hybrid approaches

Method	# of instances solved within β iterations					# of iterations	
	$\beta = 1$	$\beta \leq 10$	$\beta \leq 100$	$\beta \leq 1,000$	$\beta \leq 10,000$	Average	Max
HF	21	47	67	76	81	217.44[a]	5,149[a]
HT	53	70	84	96	97	96.86[a]	1,711[a]
HP	59	96	100	100	100	3.16	81

[a]From results obtained within 1 h

average. When MILP can very accurately estimate the demand of prototypes and the allocation of tests, it is more efficient to follow the planning decision and keep CP solving the rest of the problem.

6.2 Results of solving large instances

In this section, we apply the CP approach to solve 60 instances with 600 tests each. Since MILP is no longer applicable for solving the large instances, we compare our CP results with the previously-known solutions obtained from the heuristic and GA methods. Afterward, the hybrid approaches are applied. We further achieve optimal solutions for some instances.

6.2.1 Results of solving large instances by CP approach

In this section, we apply CP with various search schemes. We consider the test selection methods: LST, Slack, Slack-LST, and LST-Slack, together with MinId and MaxSt rules for the prototype selection. Note that we specify parameter c' in Slack-LST and LST-Slack with values 1, 2, and 3 to realize various behaviors.

Also, we apply a search algorithm named AssignAlternative in ILOG Scheduler for the resource allocation process. In our case, prototypes with unit capacity are available for execution in parallel. We can use the AssignAlternative algorithm with three possible options: SelAltRes, SelResMinGlobalSlack, and SelResMinLocalSlack.

Regarding the number of available prototypes, 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. However, in practice many tests can share the same prototype. Thus, it suffices to specify $m = \lceil n/3 \rceil$.

After solving each instance by various search schemes, we determine the best feasible solutions. Afterward, we subtract these results by one prototype to initialize the number of prototypes m for Backward Planning/MinId and Backward Planning/MaxSt. In total, we apply 21 search schemes. Each computation is limited to 1 h.

The final results are summarized in Table 4. For each instance we compare the best solutions obtained from CP with the best solutions previously solved by the heuristic and GA methods. We determine the difference between the numbers of required prototypes obtained from CP and GA methods, i.e. $m_r^{\mathrm{CP}} - m_r^{\mathrm{GA}}$. The histogram is shown in Fig. 4. In most cases, CP can decrease the prototype demand. Nevertheless, there are 2 instances where the GA method finds better solutions than CP.

Furthermore, we calculate the average number of required prototypes for these 60 instances. The GA method finds feasible solutions with an average of 74.83 prototypes. We can reduce the demand to 73.37 prototypes using CP with Slack/MinId and LST/MinId. The demand decreases further to 72.12 prototypes after applying other search schemes.

In addition, Table 5 further provides the performance analysis when we use different search schemes. Of the total 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 GA method. Also, we provide a number of instances whose solutions are equal to the best feasible results known from all methods.

Table 4 Final solutions obtained from solving 60 instances of 600 tests by GA and CP approaches

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

As we use three different search schemes in ILOG Scheduler, we select only the best results among them to present here. They can solve 44 instances and need on average about 40 min to achieve their best solutions. When applying basic search algorithms like Slack/MinId and LST/MinId, we can achieve feasible solutions for 56 and 55 instances. The Slack/MinId algorithm alone can contribute to the best solutions for 17 cases. Although neither Slack/MinId nor LST/MinId is perfectly robust to cope with all cases, both strategies can complement each other to solve all instances and achieve better solutions for 38 instances.

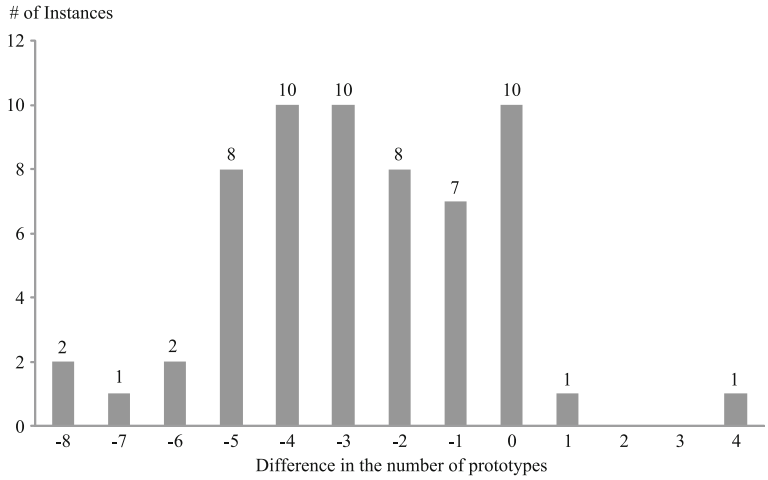

Fig. 4 Comparison of the final solutions obtained from CP and GA methods (i.e. $m_r^{CP} - m_r^{GA}$)

Table 5 Analysis of solving 60 instances of 600 tests using CP with various branching schemes

Job-selection method	Machine-selection method	Time (s)	# of instances which can be solved by CP	# of instances whose CP results better than GA	# of instances whose CP results equal best values
Scheduler		2,592.51	44	2	0
Slack	MinId	268.63	56	37	17
	MaxSt	278.08	58	39	17
LST	MinId	273.83	55	13	1
	MaxSt	263.76	55	12	1
Slack- LST1	MinId	262.70	57	35	13
	MaxSt	257.01	56	36	10
Slack-LST2	MinId	264.11	58	37	18
	MaxSt	264.68	59	38	16
Slack-LST3	MinId	264.39	58	35	18
	MaxSt	268.08	58	36	18
LST-Slack1	MinId	267.44	58	27	6
	MaxSt	265.30	58	25	6
LST-Slack2	MinId	282.48	57	27	9
	MaxSt	260.85	58	27	8
LST-Slack3	MinId	277.48	56	34	15
	MaxSt	272.47	55	32	14
Backward	MinId	66.35	18	48	57
Planning	MaxSt	69.84	16	46	56
Best results by Slack+LST/MinId			60	38	18
Best results w/o backward planning			60	46	47
Best results from CP			60	48	58

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. In the best case, Slack-LST2/MaxSt cannot solve just one instance.

Before applying the backward planning scheme, CP has already improved the solutions for 46 instances. We then use our best solutions from CP to initialize the number of prototypes available for the backward planning schemes. As a result, we additionally achieve 2 instances whose solutions become better than the GA method. In fact, the backward planning schemes find new solutions for 11 instances. Mostly, we further improve our own best results.

Regarding the computation time, CP takes around 5 min on average to achieve the best solutions. For the remaining computation time of around 1 h, CP gets lost in the huge search tree. We should actually not wait too long and rely only on backtracking. It is better to restart computation with different search schemes allowing us to explore other possibilities.

As shown by Bartels and Zimmermann [3], their heuristic method solved each instance 500 times with different priority-rules and different weight values. The total computation time was around 60 s. After that, the GA method containing 30,000 populations was applied. Their best solutions were achieved within around 3.5 min.

Table 6 Average number of distinct values of release and due dates

α	# of instances	Before reduction		After reduction	
		Avg. $\lvert R_v \rvert$	Avg. $\lvert D_v \rvert$	Avg. $\lvert R_v \rvert$	Avg. $\lvert D_v \rvert$
1	1	92	63	47	35
2	18	108	80	39	33
3	29	126	98	39	35
4	11	143	117	40	37
5	1	163	133	44	42

As we try to apply various searches, the total computation time for CP is longer but still within a reasonable range especially for the long-term planning. Moreover, CP is more convenient for new project implementation. At the beginning we just concentrate on problem formulation and apply an available default search. To improve the performance, we can later develop more efficient search strategies.

6.2.2 Results of solving large instances by hybrid approach

As mentioned in Section 5.5, we apply Algorithm 6 to neglect some values in R_v and D_v in order to reduce the size of the planning MILP model. When we start from $\alpha = 0$, the MILP models of all instances become too large. Table 6 shows the number of instances corresponding to different values of α. We notice that each variant initially has to consider around one hundred values in R_v and D_v. After using the procedure, the sizes of both sets decrease to around 40 values.

Moreover, to solve the scheduling CP model, we sequentially use the following search schemes: Slack/MinId, LST/MinId, Slack-LST2/MaxSt, LST-Slack2/MaxSt, and Backward/MinId. The results from the previous section show that Slack/MinId and LST/MinId can complement each other. The other algorithms are selected by using the number of solved instances as a criterion. The computation results in Section 6.2.1 suggest that we can set a time limit of 10 min for each CP search scheme. Also, we specify time limits of 1 h and 5 h for the MILP and overall computation.

Only HP hybrid approach can optimally solve 7 instances as shown in Table 7. For 2 instances, we obtain the same objective values found before by CP. However, CP cannot prove their optimality. The planning process can really help the scheduling CP model to achieve better solutions for other 5 instances. Remember that we previously applied CP with the same search schemes to solve the complete problem.

Table 7 Optimal solutions of solving large instances by HP hybrid approach

No.	m_r	Computational time (s)			# of iterations	CP search scheme
		MILP	CP	Total		being able to find solution
1	62	194.83	31.26	226.09	2	Slack/MinId
23	61[a]	119.05	25.8	144.85	2	Slack/MinId
24	61	142.85	27.89	170.74	2	Slack/MinId
28	61[a]	1,084.02	939.97	2,023.99	32	LST-Slack2/MaxSt
44	61[a]	302.48	628.35	930.83	6	Slack-LST2/MaxSt
54	62[a]	540.73	926.71	1,467.44	8	LST-Slack2/MaxSt
58	61[a]	78.64	23.84	102.48	1	Slack/MinId

[a]Improved solution

Without the supervision from the master MILP model, CP found solutions required more prototypes.

We notice also that no invalid cuts happen at all while solving these instances. When solving a large model, CP cannot complete within the specified time limit. However, these cuts are included because of the infeasibility detected from solving small sub-CP models.

Moreover, it is still useful to apply various search algorithms. For the additional 3 instances, Slack-LST2/MaxSt and LST-Slack2/MaxSt can find complete solutions overlooked by Slack/MinId.

Our hybrid approach fails to achieve complete solutions for other instances. For 24 instances, MILP obtains only feasible solutions, while CP cannot further achieve complete schedules. For 28 instances, MILP can optimally solve problems in every iteration. However, invalid cuts occur since we cannot prove the infeasibility of large CP models. We suppose a few more optimal solutions could be revealed if we improve our search schemes. Finally, there is one instance whose MILP model can be optimally solved. However, sub-CP models easily detect infeasibility in every iteration. We can realize that MILP cannot provide useful estimation for the scheduling process.

Notice that a stronger cut policy can be applied if we know a minimum set of jobs which causes the conflict by using the algorithm suggested by Junker [11]. 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 become smaller, we observe that many of them still cannot be solved by CP within reasonable time.

7 Conclusion

We discuss a scheduling problem in 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 of prototype production. Tests must be allocated to appropriate prototypes while observing component requirements, temporal, and problem-specific constraints. Our objective is to minimize the peak demand of prototypes required for the testing process.

First, the CP approach is applied to solve our problem. CP helps us to naturally express our complex set of requirements while allowing a compact and scalable problem formulation. Moreover, we introduce our own search schemes trying to schedule critical tests first and to use as few prototypes as possible.

Furthermore, we propose a hybrid approach based on the Benders decomposition. MILP and CP are applied to address the master planning and the slave scheduling problems. The MILP model is based on several mathematical principles: the maximum clique problem, set covering, and energetic reasoning. For the scheduling process, we introduce various CP models with different abilities to alternate the planning decision. In addition, for solving the large instances the size of the MILP model must be reduced by neglecting some values of release dates and due dates, while we consecutively apply various CP branching schemes to cover potential search areas.

To evaluate our approaches, we deploy the existing benchmark previously solved by heuristics and GA. CP can find better solutions for most of the large instances containing 600 tests. Also, our search schemes outperform the algorithms provided in our CP solver. These general search rules are still useful when we initially formulate and solve a problem. To improve the performance, we can further develop efficient search strategies based on experience of human planners and knowledge acquired from the heuristic method.

Notice that the CP method performs like an upper bound procedure since it aims to achieve a feasible solution first and tries to improve the result later. On the contrary, the hybrid method starts from a lower bound and iteratively improves to reach an optimal solution.

Although our HP hybrid method can solve most of the small instances faster than CP, it needs more computation time on average. Only for solving a few of the small instances, the hybrid method spends computational effort on a number of iterations to eliminate invalid solutions suggested by the master MILP model. Therefore, the performance of the hybrid method depends on whether the planning process can provide precise demand estimation and test allocation.

It is still challenging to tighten the master MILP model such that its size does not exceed memory capacity when we solve large instances. As the planning process cannot completely consider the impact of several side constraints, it is better to allow CP to partially reallocate the MILP decision and to search for a complete schedule. Finally, we show that our hybrid approach can optimally solve some of the large instances, most of which could not be achieved before by using CP alone.

Acknowledgements The author is grateful to Prof. Dr. Chris Beck for his advice during the Doctoral Programme in CP 2007 and Dr. Jan-Hendrik Bartels for cooperation in using his random instances. The work was supported by a grant from the NRW Graduate School of Production Engineering and Logistics.

References

1. Baptiste, P., Le Pape, C., & Nuijten, W. (2001). *Constraint-based scheduling: Applying constraint programming to scheduling problems*. Norwell, MA: Kluwer.
2. Bartels, J.-H. (2008). *Anwendung von Methoden der Ressourcenbeschränkten Projektplanung mit multiplen Ausführungsmodi in der betriebswirtschaftlichen Praxis*. PhD thesis, Clausthal University of Technology.
3. Bartels, J.-H., & Zimmermann, J. (2009). Scheduling tests in automotive R&D projects. *European Journal of Operational Research, 193*(3), 805–819.
4. Benini, L., Bertozzi, D., Guerri, A., & Milano, M. (2005). Allocation and scheduling for MPSoCs via decomposition and no-good generation. In *Principles and Practice of constraint programming—CP 2005*. Lecture notes in computer science (Vol. 3709/2005, pp. 107–121). New York: Springer.
5. Demeulemeester, E. (1995). Minimizing resource availability costs in time-limited project networks. *Management Science, 41*(10), 1590–1599.
6. Hooker, J. (2006). An integrated method for planning and scheduling to minimize tardiness. *Constraints, 11*(2), 139–157.
7. Hsu, C. C., & Kim, D. S. (2005). A new heuristic for the multi-mode resource investment problem. *Journal of the Operational Research Society, 56*(4), 406–413.
8. ILOG (2006). *ILOG CPLEX 10.0 user's manual*. ILOG S.A.
9. ILOG (2006). *ILOG scheduler 6.2 user's manual*. ILOG S.A.
10. Jain, V., & Grossmann, I. E. (2001). Algorithms for hybrid MILP/CP models for a class of optimization problems. *INFORMS Journal on Computing, 13*(4), 258–276.

11. Junker, U. (2001). Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *Proceedings of IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*.
12. Le Pape, C., Couronné, P., Vergamini, D., & Gosselin, V. (1994). Time-versus-capacity compromises in project scheduling. In *Proceedings of the thirteenth workshop of the UK planning special interest group*.
13. Li, H., & Womer, K. (2009). Scheduling projects with multi-skilled personnel by a hybrid MILP/CP benders decomposition algorithm. *Journal of Scheduling, 12*(3), 281–298.
14. Limtanyakul, K. (2009). *Scheduling of tests on vehicle prototypes*. PhD thesis, Dortmund University of Technology.
15. Limtanyakul, K., & Schwiegelshohn, U. (2007). Scheduling tests on vehicle prototypes using constraint programming. In *Proceedings of the 3rd multidisciplinary international scheduling conference: Theory and applications* (pp. 336–343).
16. Lockledge, J., Mihailidis, D., Sidelko, J., & Chelst, K. (2002). Prototype fleet optimization model. *Journal of the Operational Research Society, 53*(8), 833–841.
17. Möhring, R. H. (1984). Minimizing costs of resource requirements in project networks subject to a fixed completion time. *Operations Research, 32*(1), 89–120.
18. Neumann, K., & Zimmermann, J. (1999). Resource levelling for projects with schedule-dependent time windows. *European Journal of Operational Research, 117*(3), 591–605.
19. Neumann, K., Schwindt, C., & Zimmermann, J. (2003). *Project scheduling with time windows and scarce resources: Temporal and resource-constrained project scheduling with regular and nonregular objective functions*. New York: Springer.
20. Nübel, H. (1998). *A branch-and-bound procedure for the resource investment problem with generalized precedence constraints*. Tech. rep., Institut für Wirtschaftstheorie und Operations Research, University of Karlsruhe.
21. Pardalos, P. M., & Xue, J. (1994). The maximum clique problem. *Journal of Global Optimization, 4*(3), 301–328.
22. Pinedo, M. (2002). *Scheduling-theory: Algorithm and systems* (2nd ed.). Englewood Cliffs, NJ: Prentice Hall.
23. Scheffermann, R., Clausen, U., & Preusser, A. (2005). Test-scheduling on vehicle prototypes in the automotive industry. In *Proceedings of sixth Asia-Pacific industrial engineering and management systems (APIEMS)* (Vol. 11, pp. 1817–1830).
24. Schwindt, C. (1998). *Generation of resource-constrained project scheduling problems subject to temporal constraints*. Tech. rep. WIOR-543, Institut für Wirtschaftstheorie und Operations Research, University of Karlsruhe.
25. van Beck, P. (2006). Backtracking search algorithms. In F. Rossi, P. van Beek, & T. Walsh (Eds.), *Handbook of constraint programming* (pp. 85–134). New York: Elsevier.
26. Yamashita, D. S., Armentano, V. A., & Laguna, M. (2006). Scatter search for project scheduling with resource availability cost. *European Journal of Operational Research, 169*(2), 623–637.
27. Zakarian, A. (2010). A methodology for the performance analysis of product validation and test plans. *International Journal of Product Development, 10*(4), 369–392.