Paweł SITEK[*], Jarosław WIKAREK[**]

# APPLICATION OF DIFFERENT COMPUTATIONAL ENVIRONMENTS TO OPTIMIZATION OF MAKESPAN IN THE FLOW PRODUCTION LINE

**Abstract**

*A mathematical model of makespan optimization in the divided flow production line is presented. The methodology of Constraint Programming and Integer Programming to optimization of makespan in the above environment have been considered. Multiple examples illustrate the concept proposed*

## 1. INTRODUCTION

Shop scheduling has been researched in many varieties. The basic shop scheduling model consists of **machines** and **jobs** each of which consists of a set of **operations**. Each operation has an associated machine on which it has to be processed for a given length of time. The processing times of operations of a job cannot overlap. Each machine can process at most one operation at the given time. In the basic models are **m** machines and **n** jobs. The processing time of an operation of job **j** on machine **i** is denoted by $\mathbf{p_{ij}}$ and $\mathbf{p_{max} = max\ p_{ij}}$ **.** The three well-studied models are the open shop, flow shop and job shop problems. In an open shop problem, the operations of a job can be performed in any order. In a job shop problem the operations must be processed in a specific, job-depended order. A flow shop is a special case of a job shop in which each job has exactly m operations- one per machine. And also the order in which they must be processed is the same for all the jobs. The problem is to minimize makespan. Makespan is the overall length, of the schedule with the above constraints [1]. All above mentioned problems are strongly NP-hard. For the flow shop problem, the case where there are more than two machines is strongly NP-hard., although the two machines version is polynomial solvable [2].

In this chapter we present the problem which belongs to the flow shop production environment. There is a flow production line which can be dynamically divided into sections designated to concurrent processing different products. The production flow through machines belonging to the section is synchronized. The set of products which can be manufactured in a given line depends on tools the line is equipped with. Each feasible set of products processed concurrently in the line has been named the production variant. This is a common manufacturing environment for repetitive production in a small or medium sized manufacturing company. Moreover, we present two computational philosophies and three computational environments in illustrative examples for the above problem.

[*] Dr inż., Technical University of Kielce, e-mail: sitek@tu.kielce.pl
[**] Dr inż., Technical University of Kielce, e-mail: j.wikarek@tu.kielce.p

## 2. DESCRIPTION OF THE PROBLEM OF OPTIMIZATION OF MAKESPAN IN THE FLOW PRODUCTION LINE

In this chapter we will consider a production system (see fig.1) which consists of a flow production line, which is composed of N identical workstations (machines). The line can be dynamically divided into sections which execute operations on their products $j \in J$ and each machine (section) is equipped with tools at the same time (setup time).

The problem for this particular case is how to allocate products to production variants $s \in S$ of the line to minimize makespan and cover requirements for the system products Zj.

Better allowance of products to variants assures:

- Better utilization of machines, shortening of time of production orders execution
- Smaller numbers of production variants which involves smaller frequency of setups.

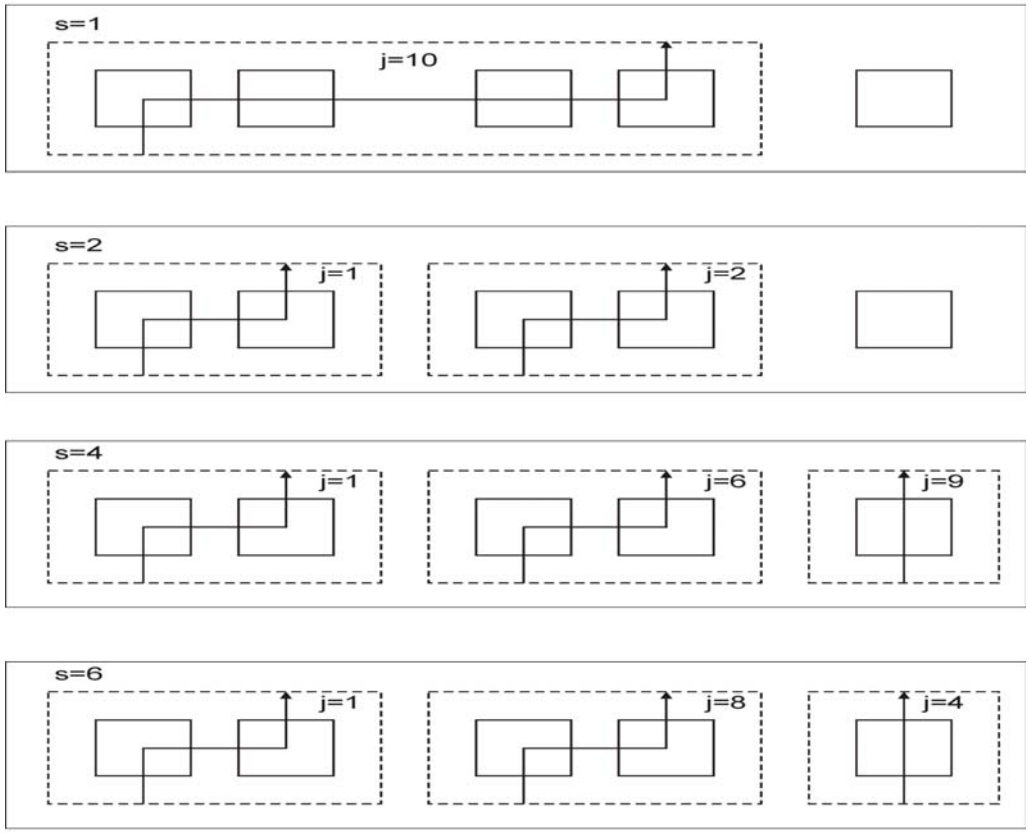


Fig.1. Flow production line in exemplified variants: s=1, s=2, s=4 and s=6. The line is divided into two sections in variant s=2 for products j=1, j=2.The line is dived into three sections for variant s=4 for products j=1,j=6, j=9 and for variant s=6 products j=1, j=8, j=4.At the end the line has one section in variant s=1 for product j=10 - (see example_1)

## 3. MATHEMATICAL MODEL OF OPTIMIZATION OF MAKESPAN IN THE FLOW PRODUCTION LINE

In the system of a flow production line (fig.1) products $j \in J$ are produced, where J is a set of all products. Total production from all production variants of the line should cover requirements for the system products. This dependence can be expressed as equations (1),

$$\sum_{s=1}^{S} z_{sj} = Z_j \qquad \text{for } j \in J, \tag{1}$$

where:

$Z_j$ – quantity of order of the product $j$,

$z_{sj}$ – planned quantity of the product $j$ from the variant s of line, $j \in J$, $s = 1..S$.

Processing the product demands definite number of operations. Every operation is executed on a single machine. It is obvious that the total number of simultaneously executed operations may not be greater than number of machines in the production line.

Thus,

$$\sum_{j \in J} x_{sj} K_j \le N, \qquad \text{for } s = 1..S, \tag{2}$$

*where*

$$x_{sj} = \begin{cases} 1, \text{if product j is produced in variant s of line,} \\ 0, \text{otherwise,} \end{cases}; \; j \in J, s = 1..S,$$

$K_j$ – number of operations of the product $j \in J$, as well as number of machines in the section producing the product,

$N$ – number of machines in the line production line.

The run time $t_s$ for the variant s of the production line may not be shorter than the run time of any product processed in this variant (fig.2).This condition can be expressed as inequality:

$$p_j z_{sj} \le t_s, \qquad \text{for } j \in J, s = 1..S, \tag{3}$$

where

$p_j$ – production pace of the product j. It is equal to the longest operation time of the product $j \in J$.

$$y_s = \begin{cases} 1, \text{if the variant h of the production line really exists,} \\ 0, \text{otherwise} \end{cases}; \; s = 1..S,$$

The binary decision variables $y_s$ are introduced to the model because the number of variants for a given line is unknown and S is its assumed upper bound. If the product j is not allocated to the variant s of the production line, that is to say when $x_{sj}=0$, then the proper quantity $z_{sj}$ should be equal to 0. Similarly, if variant s does not exist for the production line, that is to say when $y_s=0$, then its run time $t_s$ equals 0.

These rules are equivalent to conditions (4) and (5),

$$p_j z_{sj} \le x_{sj} T_d, \qquad \text{for } j \in J, s = 1..S, \tag{4}$$

$$t_s \le y_s T_d, \qquad \text{for } s = 1..S, \tag{5}$$

$$T_d \geq \sum_{j=1}^{J} (p_j Z_j + \tau) \tag{6}$$

Obviously, binary decision variables must satisfy the following constraints (7)..(11)

$$x_{sj} \in \{0,1\}, \qquad \text{for } j \in J, s = 1..S, \tag{7}$$

$$y_s \in \{0,1\} \qquad \text{for } s = 1..S, \tag{8}$$

$$y_s \geq x_{sj}, \qquad \text{for } j \in J, s = 1..S, \tag{9}$$

$$y_s \leq \sum_{j \in J} x_{sj}, \qquad \text{for } s = 1..S, \tag{10}$$

$$f: C_{\max} = \sum_{s=1}^{S} (t_s + y_s \tau), \tag{11}$$

where

$\tau$ – setup time, which is assumed, for simplicity, to be identical for all variants

The goal function of this problem is the makespan (11). The problem of optimization makespan in flow production line which can be divided into sections is to minimize *f* under constraints from (1) to (10). Thus, this is an integer programming problem.

## 4. THE COMPUTATIONAL ENVIRONMENTS FOR THE OPTIMIZATION

To solve the optimization problem three computational environments were used: package LINGO, language CHIP as well as Oz/MOZART. All are declarative environments. The LINGO supports standard operation research (OR) methods and algorithms like: Simplex Method, Successive Linear Programming (SLP), Generalized Reduced Gradient (GRG), Branch-and-Bound method.

The CHIP and Oz/MOZART support paradigm of constraint programming. Constraint programming (CP) is a declarative programming technique that has grown from the cooperation of several research disciplines including Artificial Intelligence, Computational Logic, Programming Languages and Operation Research. It has become an indispensable methodology and implementation for modeling and solving difficult combinatorial problems. Its highly declarative nature and its powerful solving methods have led to its commercial success.

### 4.1. LINGO

LINGO is a simple tool for performing complex and powerful task. It's easy to use, with smaller number commands than traditional programming languages. At the simplest level, this means that with LINGO you can solve equations with many independent variables (direct models) or with interdependent variables (simultaneous models) by entering just a few simple lines. The LINGO has modeling language, which unlike conventional programming languages

such as Pascal, Basic or C, is nonprocedural. That is, when you specify a model for LINGO to solve, you only tell you what you want, not how it should find the solution.

It's LINGO job to worry about how. In this sense LINGO is known as a specification language (similar to declarative languages). You tell it what you want and it does the rest.

LINGO's modeling language lets you express the problem in a natural manner, which is very similar to standard mathematical notation.

LINGO has four solvers:

- A direct solver,
- A simultaneous linear solver/optimizer,
- A simultaneous nonlinear solver/optimizer,
- A branch-and-bound manager for models with integer restrictions.

LINGO determines which solvers to use on a model by examining its structure and mathematical content. The basic structure of a LINGO optimization model consists of the MODEL and END statements. The rest of the text is the model. There are three optional sections: SETS, DATA and INIT.

**The SETS section of a model**

Sets are simply groups of related objects. A set might be a list of products, tasks, orders or stocks. The SETS section begins wit the word SETS: and ends with word ENDSETS.

**The DATA section of a model**

For the purpose of giving values to same set attributes before the LINGO can solve the model, the package uses a second optional section the DATA section. Similar to the SETS section begins with the word DATA: and ends with the word ENDDATA. In this section you type expressions to initialize the attributes of the sets you defined in the SETS sections. In the DATA section you can also use file import functions: @import and @file to give values to some set attributes from external files.

**The INIT section of a model**

To give initial values to attributes (usually unknown for which you are solving) LINGO uses an optional model section called the INIT section. It begins with the word INIT: and ends with the word ENDINIT.

Most templates consist of three sections:

- a structure section giving definitions of variables and SETS;
- a DATA section with input data for the particular problem;
- the model equations.

The source code of model implementation in the LINGO environment has been presented in the appendix A.

### 4.2. CHIP (Constraint Handling in Prolog)

CHIP is a constraint logic programming language (CLP) designed to tackle efficiently the so-called constrained search problems. CLP may be defined as a body of techniques used for solving problems with constraints. The essence of CLP: modeling combinatorial, continuous and mixed decision problems with the help of constraints and logical relations, solving combinatorial, continuous and mixed decision problems by analyzing and propagating the constraints [5].

The main idea of CLP is:

- Problems to be solved are modeled using elementary logic, in a way that turns the model into a part of the problem-solving program.
- Exploring constraints, which should be satisfied by the solutions, generates solutions.

CHIP is a new generation programming language which combines powerful modeling capability, flexibility and versatility of advanced Artificial Intelligence tools with the efficiency of conventional algorithmic approaches. CHIP provides different programming methods and problem solving techniques in one programming environment.

There are Logic programming, Constraint programming, Linear programming, Integer programming, Branch-and-Bound procedure, etc.

The source code of model implementation in the CHIP has been presented in the appendix B.

### 4.3. Oz/MOZART

In the CLP framework the underlying constraint solver is treated as a black box. Which cannot be changed or even looked at by the user? Moreover, the user cannot extend such constraint handler at the program level, because the language does not provide the necessary operations on constraints which are usually used by the constraint solution algorithms. There is already a language framework which seems to have the desired flexibility which is missing in CLP. It is the concurrent constraint programming (CCP) framework which can be thought of as CLP plus concurrency. More precisely a CCP program consists of set concurrent agents which share a set of variables that are subject to some constraints [7].

Oz is a new language combining functions with relations so that it has the potential for extra expressiveness in the constraint solver. Oz is a good platform to integrate algorithms from OR to achieve an amalgamation of a high-level constraint language with efficient OR methods. Oz is a CCP language for functional, object-oriented, and constraint programming (CP) [6]. The unique advantages of Oz, which can be offered to the OR problems, are:

- **Expressiveness**. Different language paradigms allow a natural and high-level modeling of the problem. Concurrent constraints provide for rapid prototyping and testing of different models.
- **Programmable Search**. The users can program their own strategies. Search is separated from the reduction of search space achieved by constraint propagation.
- **Modularity**.
- **Openness**. Through the use of C++, new constraints can be implemented efficiently by the programmer and used like any Oz procedure.

Oz provides algorithms to decide the satisfiability and implications for basic constraints which take form x=n, x=y; x, y $\in$ D where x and y are variables, n is an nonnegative integer value and D is finite domain. The basic constraints reside in the constraint store. Non-basic constraints, such x-y=z, are not contained in the constraint store but are imposed by propagators. An Oz propagator is a computational agent which is posted on the variables occurring in the corresponding domain. It reads the constraint store and tries to narrow the domains. The propagators try to do by amplifying the store with basic constraints.

*Example:*
*Constraint store containing the domain variables X, Y wit the domain [1,..5].*
*The propagator is running for constraint X+Y =4*
*The propagator narrows the domain for both variables [1..3].*
*Adding the constraint X=1 narrows the domain of X to 1 and the domain of Y to 3.*

The source code of model implementation in the Oz/MOZART environment has been presented in the appendix C.

## 5. THE ILLUSTRATIVE EXAMPLES

The mathematical model (1) .. (12) is the integer programming model (classical OR model) than can be implemented in any computational environment without loss of cohesion.

In this section, two illustrative examples are presented. The exemplified flow production line consists of five identical workstations (machines) (fig. 1). The main difference between examples is the size (tab.1).

Tab.1. The size of the example

| Example | Number of | | |
|---|---|---|---|
| | Products j | Constraints | Variables (Integers) |
| example_1 | 10 | 126 | 127 (55) |
| example_2 | 26 | 467 | 468 (216) |

In *example_1* the line should produce 10 products in the maximum 10 production variants (this is upper bound of the number of production variants which is equal to the number of products). The quantity of orders of the products in *example_1* is the following $Z_j$ ={180, 18, 14, 14, 20, 20, 18, 18, 18, 14, 20}. The other data for this example $p_j$, $K_j$ are in the table 2. The setup time is identical for all variants $\tau$=10. The result of optimization is the makespan $f$= 480. For this optimal solution the proper number of variants, run times $t_s$, planned quantities $z_{sj}$ are shown in table 2.

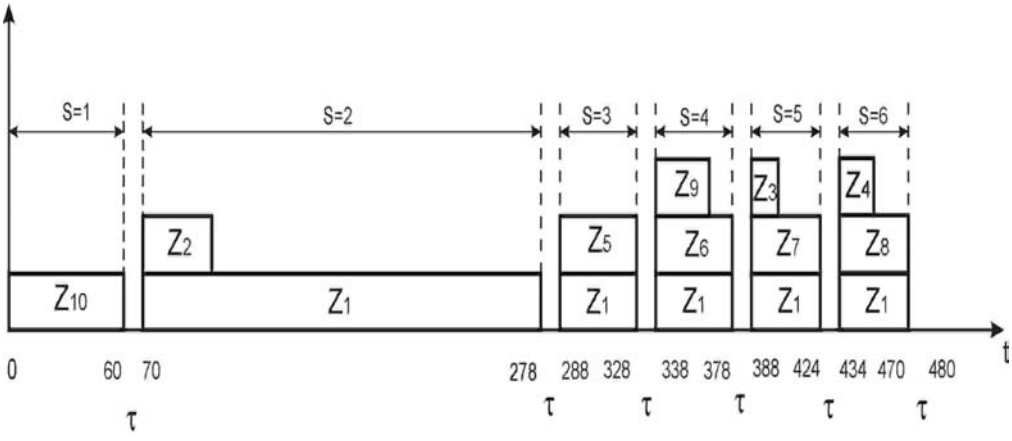


Fig.2. Gantt's chart of the optimal schedule for example_1

The Gantt's chart of the schedule for the optimal makespan $C_{max}$=480 in *example_1* is shown in fig. 2. There are six production variants. In each variant from one to three products are produced (tab.4 and fig.2).

Tab.2. Data and results for example_1

| s | | | | | $t_s$ |
|---|---|---|---|---|---|
| 1 | j | 10 | | | 60 |
| | $p_j$ | 3 | | | |
| | $z_{sj}$ | 20 | | | |
| | $K_j$ | 4 | | | |
| | $p_j * z_{sj}$ | 60 | | | |
| 2 | j | 1 | 2 | | 208 |
| | $p_j$ | 2 | 2 | | |
| | $z_{sj}$ | 104 | 18 | | |
| | $K_j$ | 2 | 2 | | |
| | $p_j * z_{sj}$ | 208 | 36 | | |
| 3 | j | 1 | 5 | | 40 |
| | $p_j$ | 2 | 2 | | |
| | $z_{sj}$ | 20 | 20 | | |
| | $K_j$ | 2 | 2 | | |
| | $p_j * z_{sj}$ | 40 | 40 | | |
| 4 | j | 1 | 6 | 9 | 40 |
| | $p_j$ | 2 | 2 | 1 | |
| | $z_{sj}$ | 20 | 20 | 14 | |
| | $K_j$ | 2 | 2 | 1 | |
| | $p_j * z_{sj}$ | 40 | 40 | 14 | |
| 5 | j | 1 | 3 | 7 | 36 |
| | $p_j$ | 2 | 1 | 2 | |
| | $z_{sj}$ | 18 | 14 | 18 | |
| | $K_j$ | 2 | 1 | 2 | |
| | $p_j * z_{sj}$ | 36 | 14 | 36 | |
| 6 | j | 1 | 4 | 8 | 36 |
| | $p_j$ | 2 | 1 | 2 | |
| | $z_{sj}$ | 18 | 14 | 18 | |
| | $K_j$ | 2 | 1 | 2 | |
| | $p_j * z_{sj}$ | 36 | 14 | 36 | |

In *example_2* the production line should produce 26 products in the maximum 26 production variants (this is upper bound of the number of production variants which is equal to the number of products). The quantity of orders of the products is the following $Z_j$ ={2,1,3,3,10,10,9,9,7,7,7,7,7,10,10,2,9,9,7,7,2,7,7,3,3,3}.The quantity of order in this example is expressed in the number of batches. The batch size carries out one hundred. The other data i.e. $p_j$, $K_j$ are in the table 2. The setup time is identical for all variants $\tau$=1. The result of optimization is the makespan $f$=126. Number of variants, run times $t_s$, planned quantities $z_{sj}$ for the optimal solution are shown in table 4.

Tab.4. Data and results for example_2

| s | | | | | | $t_s$ |
|---|---|---|---|---|---|---|
| 1 | j | 26 | 25 | 24 | 21 | 3 |
| | $p_j$ | 1 | 1 | 1 | 1 | |
| | $z_{sj}$ | 3 | 3 | 3 | 2 | |
| | $K_j$ | 1 | 1 | 1 | 1 | |
| | $p_j * z_{sj}$ | 3 | 3 | 3 | 2 | |
| 2 | j | 17 | 20 | 22 | 23 | 18 |
| | $p_j$ | 2 | 1 | 1 | 1 | |
| | $z_{sj}$ | 9 | 7 | 7 | 7 | |
| | $K_j$ | 2 | 1 | 1 | 1 | |
| | $p_j * z_{sj}$ | 18 | 7 | 7 | 7 | |
| 3 | j | 14 | 15 | 19 | | 20 |
| | $p_j$ | 2 | 2 | 1 | | |
| | $z_{sj}$ | 10 | 10 | 7 | | |
| | $K_j$ | 2 | 2 | 1 | | |
| | $p_j * z_{sj}$ | 20 | 20 | 7 | | |
| 4 | j | 13 | 18 | | | 28 |
| | $p_j$ | 4 | 1 | | | |
| | $z_{sj}$ | 7 | 9 | | | |
| | $K_j$ | 4 | 1 | | | |
| | $p_j * z_{sj}$ | 28 | 9 | | | |
| 5 | j | 10 | 11 | 12 | 16 | 7 |
| | $p_j$ | 1 | 1 | 1 | 2 | |
| | $z_{sj}$ | 7 | 7 | 7 | 2 | |
| | $K_j$ | 1 | 1 | 1 | 2 | |
| | $p_j * z_{sj}$ | 7 | 7 | 7 | 4 | |
| 6 | j | 7 | 8 | 9 | | 18 |
| | $p_j$ | 2 | 2 | 1 | | |
| | $z_{sj}$ | 9 | 9 | 7 | | |
| | $K_j$ | 2 | 2 | 1 | | |
| | $p_j * z_{sj}$ | 18 | 18 | 7 | | |
| 7 | j | 4 | 5 | 6 | | 20 |
| | $p_j$ | 1 | 2 | 2 | | |
| | $z_{sj}$ | 3 | 10 | 10 | | |
| | $K_j$ | 1 | 2 | 2 | | |
| | $p_j * z_{sj}$ | 3 | 20 | 20 | | |
| 8 | j | 1 | 2 | 3 | | 4 |
| | $p_j$ | 2 | 2 | 1 | | |
| | $z_{sj}$ | 2 | 1 | 3 | | |
| | $K_j$ | 2 | 2 | 1 | | |
| | $p_j * z_{sj}$ | 4 | 2 | 3 | | |

In both optimal solutions we can notice better utilization of machines and a smaller number of productive variants (tab. 3).

Tab.3. Number of variants and the proper utilization of machines

| Example | Number of variants (upper bound) | Number of variants (after optimization) | Utilization of machines | |
|---|---|---|---|---|
| *example_1* | 10 | 6 | s=1 | 4 |
| | | | s=2 | 4 |
| | | | s=3 | 4 |
| | | | s=4 | 5 |
| | | | s=5 | 5 |
| | | | s=6 | 5 |
| *example_2* | 26 | 8 | s=1 | 4 |
| | | | s=1 | 5 |
| | | | s=1 | 5 |
| | | | s=1 | 5 |
| | | | s=1 | 5 |
| | | | s=1 | 5 |
| | | | s=1 | 5 |
| | | | s=1 | 5 |

The considered examples of optimization of makespan in divided flow production line are problems of integer programming. Therefore the Branch-and-Bound method was applied at first to solve them. Any Branch-and-Bound (B&B) algorithm consists of two basic procedures: branching or partitioning the feasible solution into some number of subsets and bounding or estimating the optimal value of the objective (goal) function on these subsets. The B&B algorithm is implemented in many commercial and freeware computational systems. One of them is the LINGO system which consists of the modeling language and optimizer. LINGO is a simple tool for performing complex and powerful tasks [3].

For optimization of *example_1* LINGO system was used. The result of optimization $f:C_{max}$=480 was obtained. Unfortunately, it was not possible to use LINGO system for *example_2*. The calculation time was too long. Calculations were interrupted after 12 hours. So it was necessary to examine an alternative method of optimization. In view of the fact that the considered problem possesses constraints and that we have some experience in this kind of problems [4] therefore the alternative optimization method, namely the CLP (constraint logic programming) was applied. Using CHIP language we obtained the result of optimization for *example_2* (f=126).

Time of calculations in both approaches is shown in the table 5. It resulted from the version of both tools. CHIP language had to be started on the older computer (PII, 300 MHz, RAM 64 MB) under operating system DOS whereas the system LINGO was started on the computer (PIV, 1,4 GHz, RAM 512 MB) under Windows XP. Time of calculation was about one hour (see tab.3).

As standing to computational experiments environment Oz/MOZART was applied. The same values of goal function were obtained like in the case of language CHIP. However, in reasonable time it did not manage to finish calculations with confirmation of optimality of solution. Therefore in spite of obtained value one should to treat obtained solutions as feasible.

Tab.5. Time of calculation

| Example | $f$: $C_{max}$ | LINGO | CHIP | Oz/MOZART |
|---|---|---|---|---|
| Example_1 | 480 | 600s | 900s | 300s* |
| Example_2 | 126 | > 40 000s | 3900s | 300s* |

*feasible solution.

In the above computational experiments we only compared capability and efficiency of three different computational environments: LINGO – integer programming CP/CLP – constraint logic programming-CHIP and CP/CCP –constraint concurrent programming – Oz/MOZART for the particular example.

## 6. CONCLUSIONS

The methodologies of propagation of constraints with Branch-and-Bound in CP and only Branch-and-Bound in Integer Programming (LINGO) for optimization of makespan in the divided flow production line are considered. Its objective is to provide a computer-implemented model of the above presented problem. Additionally, the chapter introduces the comparison of three computational environments: CP/CLP, CP/CCP and Integer Programming in commercial solver (LINGO). There are two different computational philosophies. In the commercial solver one should transform a mathematical model to a suitable form using the language of modeling. Then solver uses the implemented algorithms and methods try to solve it. During the transformation some aspects of the problem could be lost. The idea underlying in CP is that constraints can be used to represent the problem, to solve it. In the presented problem the CP/CLP approach is more effective for the bigger size examples.

An effective CP framework was implemented in a simple flow shop production line environment for rather small companies. The extension to the whole flow shop and open shop problems is a subject of our currently conducted research. Constraint Programming framework will be implemented not only as an optimization method but as a modeling method and the method for searching feasible solutions.

Solving of NP - hard problems is always a great challenge. Using methodology which supports constraint programming to these problems is very effective. Therefore in the future one should seek environments, which comprise logic programming, constraint programming, concurrent programming and distributed programming.

## References


[1] JANIAK A.: *Wybrane problemy i algorytmy szeregowania zadań i rozdziału zasobów*. Akademicka Oficyna Wydawnicza PLJ, Warszawa, 1999.

[2] JOHNSON S.M.: *Optima two- and tree stage production schedule with setup times included*. Naval Research Logistic Quarterly, 1:61-68, 1954.

[3] *LINGO-the modeling language and optimizer*. LINDO SYSTEMS INC. Chicago, 1995.

[4] SITEK P.: *Optymalizacja uzbrojenia maszyn w systemie nadążnego sterowania produkcją.* Rozprawa doktorska, Gliwice, 2000.

[5] NIEDERLIŃSKI A.: *Constraint Logic Programming – From Prolog to CHIP*. Proceedings of the Workshop on Constraint Programming for Decision and Control, Gliwice, 1999, pp.27-34.
[6] WÜRTZ J.: *Constraint-Based Scheduling in Oz* Operations Research Proceedings 1996, 1997, Springer-Verlag.
[7] MONTANARI U., ROSSI F.: *Constraint Satisfaction, Constraint Programming and Concurrency*.

***Appendix A IMPLEMENTATION OF OPTIMIZATION MODEL IN LINGO***

```
!MODEL OF OPTIMIZATION OF MAKESPAN IN FLOW PRODUCTION LINE;
 MODEL:
  SETS:
   variants /1..9/:Ts,Ys;
   products /1..26/:j,Pj,Zj,Kj;
   allocations(variants,products):Zsj,Xsj;
  ENDSETS
  DATA:
   Pj =@file(tlocznia.ldt);
   Zj =@file(tlocznia.ldt);
   N  =@file(tlocznia.ldt);
   Kj =@file(tlocznia.ldt);
   Tal=@file(tlocznia.ldt);
   Tmax =@file(tlocznia.ldt);
  ENDDATA
!Goal function ;
 min=t;
!constraint 12;
 @sum(variants(b1):Ts(b1)+Tal*Ys(b1))=t;
!constraint 3;
 @for(allocations(b1,c1):Zsj(b1,c1)*Pj(c1)<=Ts(b1));
!constraint 1;
 @for(products(a1):@sum(variants(c1):Zsj(c1,a1))=Zj(a1));
!constraint 5;
 @for(allocations(b1,c1):Zsj(b1,c1)*Pj(c1)<=Xsj(b1,c1)*Tmax);
!constraint 6;
 @for(variants(b1):Ts(b1)<=Ys(b1)*Tmax);
!constraint 2;
 @for(variants(b1):@sum(products(c1):Xsj(b1,c1)*Kj(c1))<=N);
!constraint 8,9 (binarity of Ys, Xsj) ;
 @for(allocations(b1,c1):@bin(Xsj(b1,c1)));
 @for(variants(b1):@bin(Ys(b1)));
END
```

```
!Data for optimization example;
 !Pj= ;
2 2 1 2 2 2 2 2 1 1 1 1 4 2 2 2 2 2 1 1 1 1 1 1 1 1 1 ~
 !Zj= ;
9 9 7 7 10 10 9 9 7 7 7 7 7 10 10 9 9 7 7 7 7 7 7 3 3 3 ~
 !N= ;
5 ~
 !Kj= ;
2 2 1 1 2 2 2 2 1 1 1 1 4 2 2 2 2 2 1 1 1 1 1 1 1 1 1 ~
 !Tal=;
1 ~
 !Tmax=;
1000 ~
```

## *Appendix B IMPLEMENTATION OF OPTIMIZATION MODEL IN CHIP*

Using CHIP for solving the optimization problem (1)..(12) its constraints (1) .. (11) and the goal function (12) may be directly introduced to the problem declaration which is equivalent to the source code of the program.

```
?-[proc_tlo.pl].
?-[ogra_tlo.pl].
top:-
 wflags(72),
X   ::0..2001,   % maximum size of domain,
 Li_j::1..28,     % number of items,
 Li_h::1..30,     % number of variants,
 T   ::0..2000,   % optimization horizon,
 Txx ::0..2000,   % makespace of the system scheduling,
 Tal ::0..20,     % setup time,
 X#=150,
 write('Variable size variables'),nl,
  stale(Li_j,Li_h,T,Tal),
   write('   Number of items                :'),
   write(Li_j),nl,
   write('   Number OF variants             :'),
   write(Li_h),nl,
   write('   Optimization horizon           :'),
   write(T),nl,
   write('   Setup time                     :'),
   write(Tal),nl,
write('Laoad coefficients'),nl,
  li_t(Pj,[],Li_j,X),
  czytaj('!dane\pj.txt',Pj,Li_j),
  li_t(Zj,[],Li_j,X),
  czytaj('!dane\zj.txt',Zj,Li_j),
  li_t(Lg,[],1,X),
  czytaj('!dane\lg.txt',Lg,Li_g),
  li_t(Kj,[],Li_j,X),
  czytaj('!dane\kj.txt',Kj,Li_j),
write('Creating list of variables'),nl,
  li_t(Yh,[],Li_h,1),
  li_t(Xjh,[],L_h*Li_j,1),
  li_t(Xhj,[],Li_h*Li_j,1),
  li_t(Zjh,[],Li_h*Li_j,X),
  li_t(Th,[],Li_h,X),
write('Introducing constraints'), nl,
  og_4(Zj,Zjh,Li_h,Li_j),
  og_2(Tgh,Ygh,Txx,Li_h,Tal,Li_g),
  og_7(Xhj,Lg,Kj,Li_h,Li_j,Li_h,1),
og_5(T,Pj,Th,Th,Yh,Yh,Zjh,Xjh,Xhj,Xhj,Li_h,Li_j,Li_h*Li_j,1,1,0,1),
write('Labeling of variables     '),nl,
 przepisz(Po_1,Yh,Li_h,Po_2),
 przepisz(Po_2,Xjh,Li_h*Li_j,Po_3),
 przepisz(Po_3,Th,Li_h,Po_4),
 p1(Po_4,Txx),
 min_max(labeling(Po_1,0,most_constrained,indomain),Txx,0,700),
write('Saving results          '),nl,
  zapisz('!wyni\xhj.txt',Xhj,Li_h*Li_j,Li_j),
  zapisz('!wyni\xjh.txt',Xjh,Li_h*Li_j,L_h),
  zapisz('!wyni\zjh.txt',Zjh,Li_h*Li_j,L_h),
  zapisz('!wyni\Th.txt' ,Th,Li_h,Li_h),
  zapisz('!wyni\Yh.txt' ,Yh,Li_h,Li_h),
  zapis1('!wyni\czas.txt',Txx),nl,
```

```
nl.

stale(L_j,L_h,T,Tal):-
 open('!dane\Stale.txt',S,'r'),
   read(S,Pomo),
   L_j#=Pomo,
   read(S,Pomo1),
   L_h#=Pomo1,
   read(S,Pom23),
   T#=Pomo2,
   read(S,Pomo3),
   Tal#=Pomo3,
   close(S).

li_t(L,L,0,0).
% parameters : 1 - name of the list 2 - start list 3 - length of the list X -
size of the domain
 li_t([Zm|R],L,Nr,X):-
   Zm::0..X,
   Nr_1 is Nr-1,
   if Nr_1#=0 then
    li_t(R,L,Nr_1,0)
   else
    li_t(R,L,Nr_1,X).

czytaj(Plik,Lista,D):-
% Read "D" date from file name "Plik" to the list "Lista",
 open(Plik,S,'r'),
 li_c(Lista,[],D,S),
 close(S).

 li_c(L,L,0,0).
  li_c([Zm|R],L,Nr,S):-
  read(S,Pomo),
  Zm#=Pomo,
  Nr_1 is Nr-1,
  if Nr_1#=0 then
   li_c(R,L,Nr_1,0)
  else
   li_c(R,L,Nr_1,S).

li_d(L,L,0,0,0).
%  Display list „L” on the screen
 li_d([Zm|R],L,Nr,Zakres,Licznik):-
 L1 is Licznik+1,
 write(Zm),
 write(' '),
 if L1 #= Zakres then
  (
   writeln(' '),
   write('     '),
    L2 is 0
  )
 else
   L2 is L1,
   Nr_1 is Nr-1,
   if Nr_1 #>0 then
    li_d(R,L,Nr_1,Zakres,L2).

przepisz([A|B],[C|D],Zakres,Po_x):-
 L1 is Zakres -1 ,
```

```
 A = C,
 if L1 #>0 then
   przepisz(B,D,L1,Po_x)
 else
   Po_x = B.

p1([A|B],Xxx):-
 A = Xxx,
 Xxx#>=0,
 B= [] .

zapisz(Plik,Lista,D,Linia):-
% Write „D” date to file  "Plik" from list "Lista",
  open(Plik,S,'w'),
  li_z(Lista,[],D,S,Linia,0),
  close(S).

 li_z(L,L,0,0,0,0).
  li_z(L,L,0,0,0,0).
  li_z([Zm|R],L,Nr,S,Linia,Licznik):-
  L1 is Licznik+1,
  write(S,Zm),
  write(S,' '),
  if L1 #= Linia then
   (
    writeln(S,' '),
    L2 is 0
   )
  else
    L2 is L1,
  Nr_1 is Nr-1,
  if Nr_1#=0 then
   li_z(R,L,Nr_1,0,0,0)
  else
   li_z(R,L,Nr_1,S,Linia,L2).

zapis1(Plik,Dana):-
open(Plik,S,'w'),
 write(S,Dana),
 close(S).
%---- Constrain 12 ----------------------------------
%Parameters
% 1 - [A|B]  - Th,
% 2 - Yh,
% 3 - Txx,
% 4 - Li_h,
% 5 - Tal,
og_2([A|B],Yh,Txx,Li_h,Tal):-
 s2([A|B],Yh,Txx,Tal,0,Wynik,Li_h,Po_1,Po_2),
  Txx #>= Wynik.

  s2([],[],_,S,S,_,[],[]).
  s2([A|B],[C|D],Txx,Tal,S,R,Dlugo,Po_1,Po_2):-
   Dl1    is Dlugo-1,
   Txx #>= A + Tal * C + S,
   if Dl1 #=0 then
    (
     s2([],[],_,S+A+Tal*C,R,_,[],[]),
     Po_1  = B,
     Po_2  = D
    )
```

```
   else
    s2(B,D,Txx,Tal,S+A+Tal*C,R,Dl1,Po_1,Po_2).

%---- Constrain 1---------------------------------
%Parametry,
% 1 - [A|B]  - Zj,
% 2 - Zjh,
% 3 - Li_h,
% 4 - Licz   - current processing item,
og_4([A|B],Zjh,Li_h,Licz):-
   N is Licz - 1,
   sum_4(A,Zjh,0,Wynik,Li_h,Po_1),
   Wynik #= A,
   if N #> 0 then
    og_4(B,Po_1,Li_h,N).

sum_4(_,[],S,S,_,[]).
 sum_4(A,[H|T],S,R,Wszystkie,Po_1):-
       W         is Wszystkie-1,
       A  #>= H + S, % Zlecenie wieksze od czesci
       if W #=0 then
        (
         sum_4(_,[],H+S,R,_,[]),
         Po_1 = T
        )
       else
         sum_4(A,T,H+S,R,W,Po_1).

%---- Constrain 3,5,6,11---------------------------------
% 0   - T       -  czas optymalizacji
% 1   - [A1|B1] -  Pj,
% 2   - [C1|D1] -  Th,
% 3   - Th,
% 4   - [X|Y]   -  Yh,
% 5   - Yh,
% 6   - [A|B]   -  Zjh,
% 7   - [C|D]   -  Xjh,
% 8   - [E|F]   -  Xhj,
% 9   - Xghj    -  Xhj,
% 10  - Li_j,
% 11  - Li_h,
% 12  - Li_j,
% 13  - Licz
% 14  - War,
% 15  - Kolumna
og_5(T,[A1|B1],[C1|D1],Th,[X|Y],Yh,[A|B],[C|D],[E|F],Xhj,Li_h,Li_j,Licz,War,Kol
umna,St,St1):-
     L1    is Licz - 1,
     Ktory is Li_j*St+St1*Kolumna,
     szukaj(T,C1,A1,X,A,C,[E|F],Ktory,Po_2),
     if War #< L_gh then
      (
       W1   is War+1,
       Kol  is Kolumna,
       S1   is 1,
       S2   is 0,
       Po_3 = Po_2,
       Po_4 = Y,
       Po_5 = [A1|B1],
       Po_6 = D1
      )
```

```
     else
      (
       W1    is 1,
       Kol   is Kolumna +1,
       S1    is 0,
       S2    is 1,
       Po_3  = Xghj,
       Po_4  = Ygh,
       Po_5  = B1,
       Po_6  = Tgh
      ),
     if L1 #> 0 then
      og_5(T,Po_5,Po_6,Tgh,
           Po_4,Ygh,B,D,Po_3,Xghj,L_gh,Li_j,L1,W1,Kol,S1,S2).

    szukaj(T,C1,A1,X,A,C,[E|F],Ktory,Po_2):-
     K1 is Ktory - 1,
     if K1 #> 0 then
      szukaj(T,C1,A1,X,A,C,F,K1,Po_2)
     else
      (
        E        = C,          %     Xjh =  Xhj,
        C1     #>= A1 * A,     %     Th  >= Pj*Zjh,   (3),
        X * T  #>= C1,         %     Yh  =  0 Th  = 0 (6),
        X      #>= E,          %     Yh  =  0 Xhj = 0 (11),
        X      #>= C,          %     Yh  =  0 Xjh = 0 (11),
        C * T  #>= A,          %     Xjh =  0 Zjh = 0 (5),
        E * T  #>= A,          %     Xhj =  0 Zjh = 0,(5),
        Po_2     = F
      ).
%---- Constrain 2 ---------------------------------
%Parametry
% 1 - Xhj,
% 2 - [A|B]  - N,
% 3 - Kj,
% 4 - Li_h,
% 5 - Li_j,
% 6 - Obe,
% 7 - Licz,

      og_7(Xghj,[A|B],Kj,Li_h,Li_j,Obe,Licz):-
       Ob1 is Obe - 1,
       s7(Xhj,Kj,0,Wynik,Li_j,Po_1),
       Wynik #<= A,
       if Licz #< Li_h then
        (
         Li1   is Licz+1,
         Po_2  =  [A|B]
        )
       else
        (
          Li1  is 1,
          Po_2 =  B
        ),
        if Ob1 #> 0 then
         og_7(Po_1,Po_2,Kj,Li_h,Li_j,Ob1,Li1).
  s7([],[],S,S,_,[]).
  s7([A|B],[C|D],S,R,Dlugo,Po_1):-
   Dl1    is Dlugo-1,
   if Dl1 #=0 then
    (
```

```
     s7([],[],S+A*C,R,_,[]),
     Po_1  = B
    )
   else
    s7(B,D,S+A*C,R,Dl1,Po_1).
```

## Appendix C IMPLEMENTATION OF OPTIMIZATION MODEL IN OZ/MOZART

```
declare W
fun {W }
 proc {$ R}
  S X Z T Y Zj Pj Kj N Xo Ta Je Cz
  in
  S={List.make 5}
  S=[
     26             % number of items
     9              % number of variants
     5              % number of machines (N)
     150            % max time
     1              % set up times
    ]
  % Structure of data
    X={List.make  {List.nth S 1}*{List.nth S 2}}
    Z={List.make  {List.nth S 1}*{List.nth S 2}}
    T={List.make  {List.nth S 2}}
    Y={List.make  {List.nth S 2}}
    Zj={List.make {List.nth S 1}}
    Pj={List.make {List.nth S 1}}
    Kj={List.make {List.nth S 1}}
    N ={List.make 1}
    Xo={List.make {List.nth S 1}*{List.nth S 2}}
    Ta={List.make {List.nth S 2}}
    Je={List.make {List.nth S 2}}
    Cz={List.make 1}
  % Data for optimizations problem
Zj=[ 9 9 7 7 10 10 9 9 7 7 7 7 7 10 10 9 9 7 7 7 7 7 3 3 3 ]
Pj=[ 2 2 1 2 2 2 2 2 1 1 1 1 4 2 2 2 2 1 1 1 1 1 1 1 1 ]
Kj=[ 2 2 1 1 2 2 2 2 2 1 1 1 1 4 2 2 2 2 1 1 1 1 1 1 1 1 ]
N =[{List.nth S 3}]
  % Auxiliary lists
    {For 1 {List.nth S 2} 1
     proc {$ I}
      {List.nth Ta I}={List.nth S 5}
      {List.nth Je I}=1
     end
    }
  % Establishing of sizes of domains
    X  :::0#1
    Xo :::0#5
    Y  :::0#1
    Z  :::0#10
    T  :::0#30
    Cz :::0#150
    R=r(var_X:X var_Z:Z var_y:Y var_t:T time:Cz)
  % Additional calculation
    {For 1 {List.nth S 2} 1
     proc {$ J}
     {For 1 {List.nth S 1 } 1
```

```
     proc {$ K}
        % Xsj=Kj*Xojs
         {List.nth Kj K}*{List.nth X (K-1)*{List.nth S 2}+J}
          =:
         {List.nth Xo (J-1)*{List.nth S 1}+K}
        % if  Zsj=0 then Xsj=0
         {List.nth Z (K-1)*{List.nth S 2}+J}
          >=:
         {List.nth X (K-1)*{List.nth S 2}+J}
        %If Xsj=0 then Zsj=0
         {List.nth S 4}*{List.nth X (K-1)*{List.nth S 2}+J}
          >=:
         {List.nth Z (K-1)*{List.nth S 2}+J}
     end
    }
    end
   }
 % constraint 1
   {For 1 {List.nth S 1} 1
    proc{$ I}
     {FD.sum
{List.take {List.drop Z (I-1)*{List.nth S 2}} {List.nth S 2}}
       '=:'
      {List.nth Zj I}
     }
 % additional constrain 1
   {FD.atLeast
    1
{List.take {List.drop X (I-1)*{List.nth S 2}} {List.nth S 2}}
    1
   }
 % additional constrain 2
   {FD.atMost
    1
{List.take {List.drop X (I-1)*{List.nth S 2}} {List.nth S 2}}
     1
    }
    end
   }
 % constrain  2
   {For 1 {List.nth S 2} 1
    proc {$ J}
    {FD.sum
{List.take {List.drop Xo (J-1)*{List.nth S 1}} {List.nth S 1}}
      '=<:'
     {List.nth N 1}
    }
    end
   }
 % constrain 6 and additional constrain 3,4

   {For 1 {List.nth S 2} 1
    proc {$ I}
    {List.nth T I} =<: {List.nth Y I}*{List.nth S 4}
    {List.nth T I} >=: {List.nth Y I}
    {List.nth T I} =<: {List.nth Cz 1}
    end
   }
 % constrain 3 and 5
   {For 1 {List.nth S 1} 1
    proc {$ I}
```

```
     {For 1 {List.nth S 2} 1
      proc {$ K}
       {List.nth Pj I}*{List.nth Z (I-1)*{List.nth S 2}+K}
         =<:
       {List.nth T K}
       {List.nth Pj I}*{List.nth Z (I-1)*{List.nth S 2}+K}
        =<:
       {List.nth S  4}*{List.nth X (I-1)*{List.nth S 2}+K}
      end
     }
     end
    }
  % goal function
    {FD.sumC
     { Append Je {Append Ta [~1]}}
     { Append T  {Append Y  Cz  }}
      '=:'
      0
     }
  {FD.distribute generic( order:min value:min ) {Append X Cz}}
    {Browse Cz}
    {Browse T}
  end
end
Wynik={SearchBest {W}
  proc{$ Old New}
  {List.nth Old.czas 1} >: {List.nth New.czas 1}
  end
      }
{Browse Wynik}
```