12

Optimization and Scheduling Toolbox

Michal Kutil, Přemysl Šůcha, Roman Čapek and Zdeněk Hanzálek
Czech Technical University in Prague
Czech Republic

1. Introduction

TORSCHE (Time Optimization of Resources, SCHEduling) Scheduling Toolbox for Matlab is a freely (GNU GPL) available toolbox developed at the Czech Technical University in Prague. The toolbox is designed for researches in operational research or industrial engineering and for undergraduate courses. The current version of the toolbox covers the following areas: scheduling on monoprocessor/dedicated processors/parallel processors, open shop/flow shop/job shop scheduling, cyclic scheduling and real-time scheduling. Furthermore, particular attention is dedicated to graphs and graph algorithms due to their large interconnection with the scheduling theory. The toolbox offers a transparent representation of the scheduling/graph problems, various scheduling/graph algorithms, a useful graphical editor of graphs, interfaces for mathematical solvers (Integer Linear Programming, satisfiability of the boolean expression) and an interface to a MATLAB/Simulink based simulator and a visualization tool. The scheduling problems and algorithms are categorized by notation $(\alpha|\beta|\gamma)$ proposed by Graham and Błażewicz (Blazewicz et al., 1983). This notation, widely used in the scheduling community, greatly facilitates the presentation and discussion of scheduling problems.

The toolbox is supplemented by several examples of real applications, e.g. the scheduling of Digital Signal Processing (DSP) algorithms on a hardware architecture with pipelined arithmetic units, scheduling the movements of hoists in a manufacturing environment and scheduling of light controlled intersections in urban traffic. The toolbox is equipped with sets of benchmarks from the research community (e.g. DSP algorithms, the Quadratic Assignment Problem). TORSCHE is an open-source tool available at (`http://rtime.felk.cvut.cz/scheduling-toolbox/`)

In the off-line scheduling area, some tools for the development of scheduling algorithms already exist. The term *off-line* scheduling means all parameters of the scheduling problem are known a priori (Pinedo, 2008). A scheduling system developed at the Stern School of Business is called LEKIN (Pinedo et al., 2002). It was created as an educational tool and it provides six basic workspace environments: single machine, parallel machines, flow shop, flexible flow shop, job shop, and flexible job shop. Another tool is LiSA (Andresen et al., 2003). It is a software-package for entering, editing and solving off-line scheduling problems while the main focus is on shop-scheduling and one-machine problems. The graphical user interface is written in Tcl/Tk for machine and operating system independence. All algorithms are implemented externally while the parameters are passed through files. The commercial tool ILOG Scheduler from the software package ILOG CP Optimizer (ILOG, 2009) is based on

constraints programming. It provides extensions for scheduling problems in manufacturing, transportation and workforce scheduling.
There are more tools for on-line scheduling, where *on-line* means the parameters of the tasks become known on the task arrival/occur. One example is the MAST tool (Gonzalez et al., 2008) built to mainly support the timing analysis of real-time applications. A tool with close relationship to this category is TrueTime (Andersson et al., 2005), which is a Matlab/Simulink based discrete-events simulator mainly focused on real-time control systems.
The TORSCHE toolbox is mostly based on the existing well-known scheduling algorithms, but some of them were developed by our group. It is a very convenient platform for sharing ideas and algorithms among researchers and students. The toolbox has become part of a textbook for courses in scheduling (Pinedo, 2008). Our objective for developing TORSCHE was to fill the gap in the available tools for scheduling and optimization.
This chapter is organized as follows: Section 2 outlines the toolbox architecture. Section 3 summarizes the selected scheduling algorithms from the toolbox, with their practical applications. The first two algorithms are selected from the scheduling for the digital signal processing area. The first algorithm uses the problem formulation by satisfiability of the boolean expressions (SAT). The second one solves a cyclic scheduling problem by Integer Linear Programming (ILP). Moreover, the subsection demonstrates how the results of the scheduling can be simulated in Matlab Simulink using TrueTime. The third algorithm shows an original application of the minimum cost multi-commodity flow problem on the scheduling of light controlled intersections in urban traffic. The last application is focused on the graphic visualization of the scheduling results based on the Matlab Virtual Reality toolbox demonstrated on the hoist scheduling problem. Section 4 concludes the chapter.

2. Tool Architecture and Basic Notation

TORSCHE Scheduling Toolbox is written in the Matlab object oriented programming language (backward compatible with Matlab environment version 2007) and it is used in the Matlab environment as a toolbox. The toolbox includes two complementary parts. The first one is intended for solving problems from scheduling theory. Problems from this area or their parts can, very often, be reformulated to another problem which can be directly solved by a graph algorithm. For this purpose the second part of the toolbox is dedicated to graph theory algorithms.

2.1 Scheduling Part

The main classes of the scheduling part are *Task*, *PTask*, *Taskset* and *Problem*. The UML class diagram illustrating the class relationships is shown in Fig. 1. A task represented by the class of the same name is a unit of work to be scheduled on the given set of processors. The class includes task parameters as processing time, release date, deadline, etc. The instance of the class (variable `T1` depicted below) is returned by the constructor method, which has the following syntax rule:

```
T1 = task([Name,]ProcTime[,ReleaseTime[,Deadline ...
          [,DueDate[,Weight[,Processor]]]]])
```

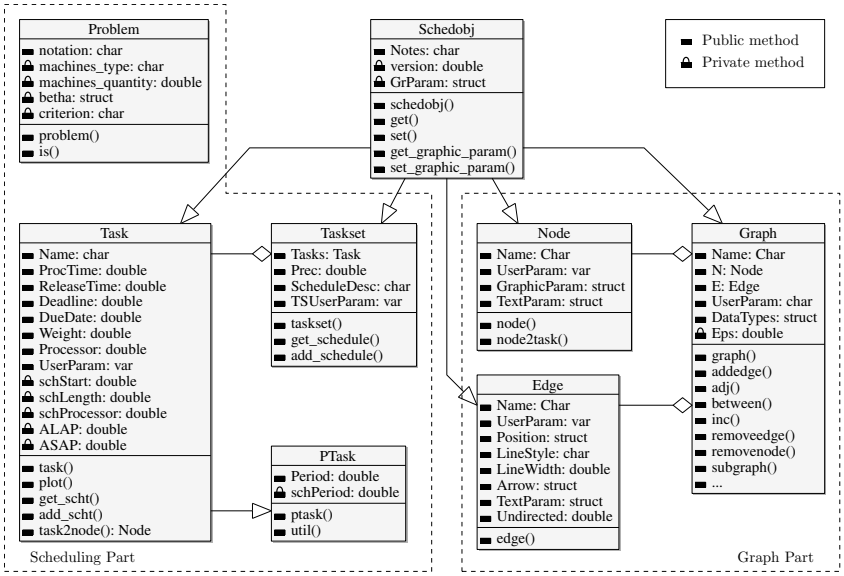


Fig. 1. TORSCHE architecture by the UML Class Diagram

Input variables correspond to the public class properties. Variables contained inside the square brackets are optional. The class *Task* provides the following properties (also graphically depicted in Fig. 2):

Processing time (`ProcTime`, p_j) is time necessary for task execution (also called computation time).

Release date (`ReleaseTime`, r_j) is the moment at which a task becomes ready for execution (also called the arrival time, ready time, request time).

Deadline (`Deadline`, $\widetilde{d}_j$) specifies a time limit by which the task has to be completed, otherwise the scheduling is assumed to fail.

Due date (`Duedate`, d_j) specifies a time limit by which the task should be completed, otherwise the criterion function is charged by a penalty.

Weight (`Weight`) expresses the priority of the task with respect to other tasks (also called the priority).

Processor (`Processor`) specifies the dedicated processors on which the task must be executed.

The resulting schedule is represented by the following properties:

Start time (`schStart`, s_j) is the time when the execution of the task is started.

Completion time (c_j) is the time when the execution of the task is finished.

Lateness $L_j = c_j - d_j$.

Tardiness $D_j = \max\{c_j - d_j, 0\}$.

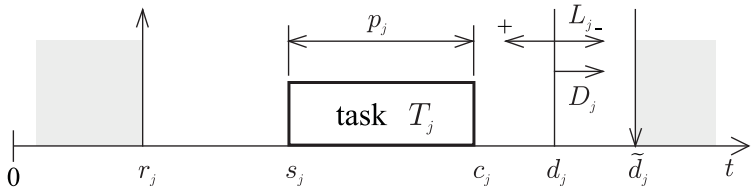


Fig. 2. Graphic representation of the task parameters

The private properties of the class *Task* are mainly intended for the final task schedule representation, which are set up inside the scheduling algorithms (e.g. by method `add_scht`). The values from the private properties are used, for example, by the method `plot` for the Gantt chart drawing.
Class *PTask* (see Fig. 1) is a derived class from the *Task* class in order to represent a periodic task in on-line scheduling problems (e.g. in response-time analysis). This class extends the *Task* class with support to store, plot and analyze the utilization methods.
The instances of the classes *Task* and *PTask* can be aggregated into a set of tasks. A set of tasks is represented by the class *Taskset* which can be obtained as the return value of the constructor `taskset`, for example:

```
TS = taskset(tasks[,prec])
```

where the variable `tasks` is an array of instances of the *Task* class. Furthermore, the relations between the tasks can be defined by precedence constraints in the optional parameter `prec`. The parameter `prec` is an adjacency matrix defining a graph where the nodes correspond to the tasks and the edges are precedence constraints between these tasks. For simple scheduling problems, the object *Taskset* can be directly created from a vector of the tasks processing times. In this case, the tasks are created automatically inside the object constructor. There are also other ways how to create an instance of the set of tasks in order to simplify the user interface as much as possible.
Another class, *Problem*, is used for the classification of deterministic scheduling problems in Graham and Błażewicz notation (Blazewicz et al., 1983). This notation consists of three parts $(\alpha|\beta|\gamma)$. The first part describes the processor environment (e.g. number and type of processors), the second part describes the task characteristics of the scheduling problem (e.g. precedence constrains, release time). The last part denotes the optimality criterion (e.g. schedule makespan minimization). The following example shows the notation string used as an input to the class constructor:

```
prob = problem('P|prec|Cmax')
```

This instance of the class *Problem* represents the scheduling problem on parallel identical processors where the tasks have precedence constraints and the objective is to minimize the schedule makespan.
All of the above-mentioned classes are designed to be maximally effective for users and developers of scheduling algorithms. The toolbox includes dozens of scheduling algorithms which are stored as Matlab functions with at least two input parameters and at least one output parameter. The first input parameter has to be an instance of the *Taskset* class containing the tasks to be scheduled. The second one has to be an instance of the *Problem* class describing

the required scheduling problem. The output parameter is an instance of the *Taskset* class containing the resulting schedule. A typical syntax of the scheduling algorithm call is:

```
TSout = algorithmname(TS,problem[,processors[,parameters]])
```

where:

`TSout`	is the instance of the *Taskset* with the resulting schedule,
`algorithmname`	is the algorithm name,
`TS`	is the instance of the *Taskset* to be scheduled,
`problem`	is the instance of the *Problem* class,
`processors`	is the number of processors to be used,
`parameters`	denotes additional parameters, e.g. algorithm strategy, etc.

The typical workflow of a scheduling problem solution is shown in a UML Interaction Overview Diagram (see Fig. 3). There are several sequence diagrams (sd) used. The first two "Create Taskset 1" and "Create Taskset 2" show the constitution of a *Taskset* instance by both of the above described ways. The third one, called "Classification", shows the constitution of a *Problem* instance. The following sequence diagram "Scheduling" presents the call of the scheduling algorithm. The scheduling algorithm is described separately in the "Scheduling Algorithm" diagram, which is divided into three parts. The first one is checking of the input parameters ("Read Properties"). The second one is constituted by the solver of a scheduling algorithm and the final part stores the resulting schedule into the instance of the *Taskset* ("Schedule to the Tasks"). The last diagram "Gantt Chart" presents the final schedule conversion to a Gantt chart, i.e. the graphical representation of a schedule.
Furthermore, the toolbox contains objects to handle problems like open shop, flow shop and job shop, it also supports limited buffers and transport robots. For more details please see the toolbox documentation (Kutil et al., 2007).

2.2 Graph Part

A number of scheduling problems can be solved with the assistance of the graph theory. Therefore, the second part of the toolbox is aimed at graph theory algorithms. All algorithms are available as a method of the main class *Graph* which is used to describe the directed graph. There are several different ways to create an instance of the class *Graph*. The graph is generally described by an adjacency matrix. In this case, the *Graph* object is created by the command with the following syntax:

```
G = graph('adj',A)
```

where the variable *A* is an adjacency matrix. Similarly, it is possible to create the *Graph* by an incidence matrix. Another way how to create the *Graph* object is based on a matrix of edge weights.
The toolbox is equipped with a simple but powerful editor of graphs called *Graphedit* based on the System Handle Graphics of Matlab. It provides a simple and intuitive way to construct directed graphs with various user parameters on nodes and edges. (see Fig. 4). The constructed graph can be easily used to create an instance of the class *Graph* which can be exported to the Matlab workspace or saved to a binary mat-file. In addition, *Graphedit* contains a system of plug-ins which allow one to extend its functionality by the user.

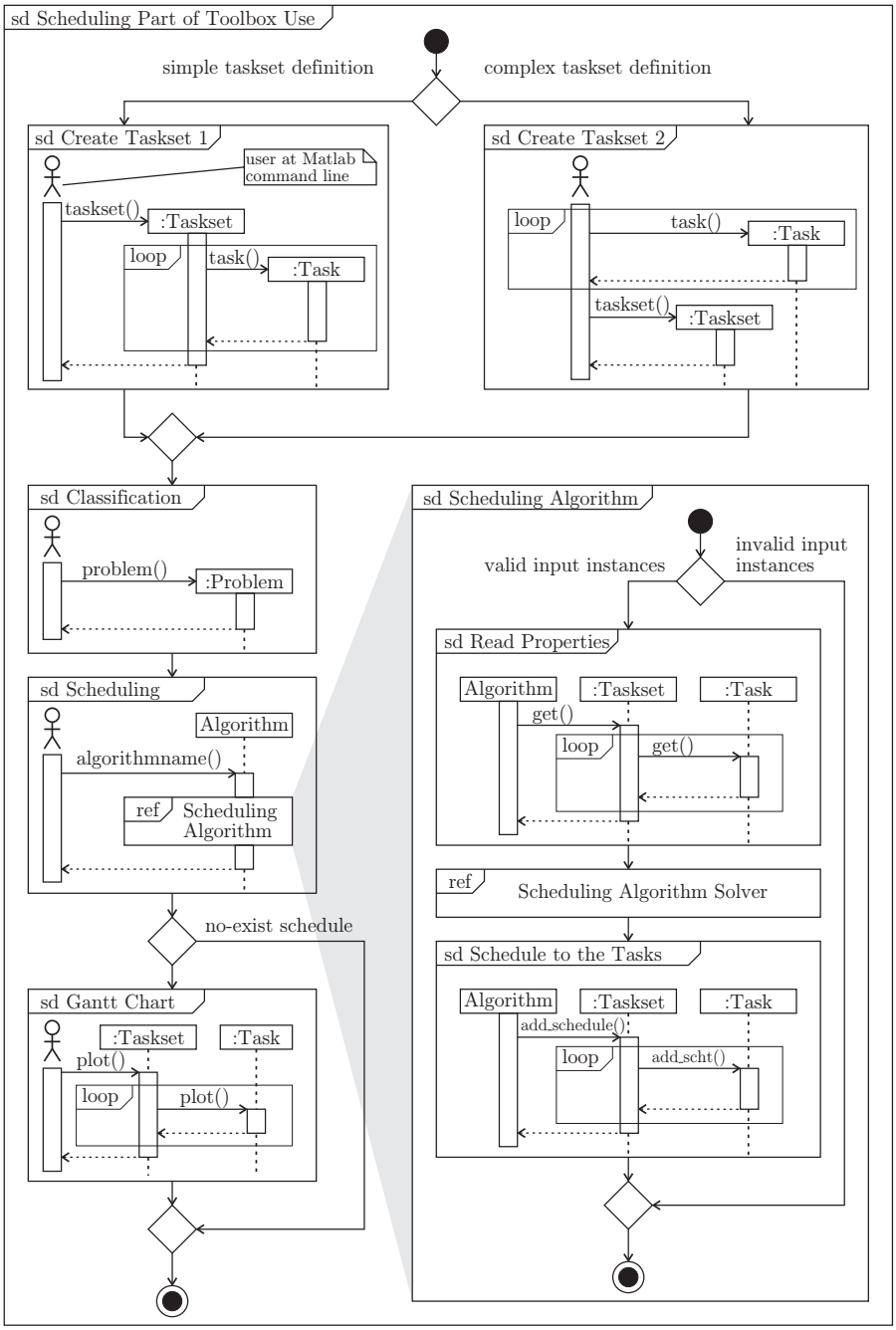


Fig. 3. UML Interaction Overview Diagram of a typical toolbox workflow of the scheduling problem solution

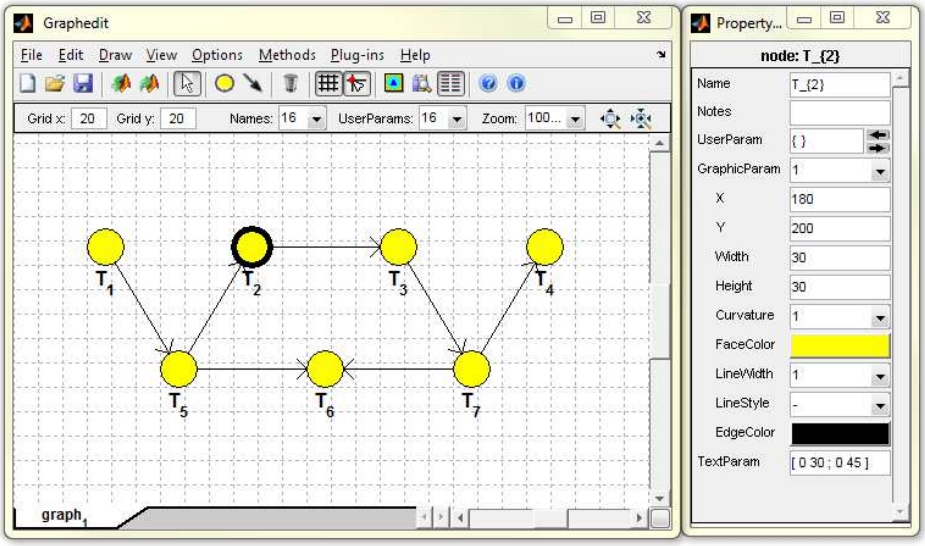


Fig. 4. Graphedit

Moreover, due to the close relationship between the scheduling and graph algorithms, each object *Graph* can be transformed to an object *Taskset* and vice versa. Obviously, the nodes from the graph are transformed to the tasks in the *Taskset* and the edges are transformed to the precedence constraints and vice versa according to the user specification.

3. Implemented algorithms

The biggest part of the toolbox is constituted by scheduling algorithms. There are a large variety of algorithms solving both simple problems (on a single processor) and practically oriented issues. This section shows several of them demonstrated on real applications.

3.1 List Scheduling Algorithm

List Scheduling (LS) is a basic and popular heuristic algorithm applicable on scheduling of set of tasks on a single and parallel processors as well. In this algorithm the tasks are ordered in a pre-specified list defining their priority. Whenever a processor becomes idle, the first available task on the list is scheduled and removed from the list. Where the availability of a task means that the task has been released and if there are precedence constraints, all its predecessors have already been processed (Leung, 2004). The algorithm terminates when all tasks from the list are scheduled. Its time complexity is $\mathcal{O}(n)$.

The accuracy of the algorithm depends on the order in which tasks appear on the list. There are many strategies defining the order of tasks in the list, e.g. the Earliest Starting Time first (EST), the Earliest Completion Time first (ECT), the Longest Processing Time first (LPT), the Shortest Processing Time first (SPT) etc. The appropriate choose of the strategy depends on the particular scheduling instance.

The List Scheduling algorithm is implemented in TORSCHE Scheduling Toolbox under function `listsch` which also allows to choice one of the implemented strategy defining the order

of tasks. Furthermore, the algorithm steps can be displayed in the MABLAB workspace by enabling the verbose mode. Moreover, the last algorithm version is able to solve scheduling problems on unrelated parallel processors. The syntax of `listsch` function is:

```
TS = listsch(T, problem, processors[, strategy][, verbose])
TS = listsch(T, problem, processors[, option])
```

where:

`T`	is the instance of the *Taskset* class without schedule,
`TS`	is the instance of the *Taskset* class with schedule,
`problem`	is the instance of the *Problem* class,
`processors`	is the number of processors,
`strategy`	is the strategy (like LPT, SPT, EST, ...),
`verbose`	is a level of verbosity,
`option`	is the optimization option setting.

This subsection concludes by an example. The example solves a problem of a chair manufacturing by two workers (cabinetmakers). Their goal is to make four legs, one seat and one backrest of the chair and assembly all of these parts within minimal time. Release time of the task representing the backrest making is equal to 20 time units. Moreover, the assemblage is divided into two stages ($\text{assembly}_{1/2}$ and $\text{assembly}_{2/2}$). Fig. 5 shows the representation of the mentioned problem instance by a graph.

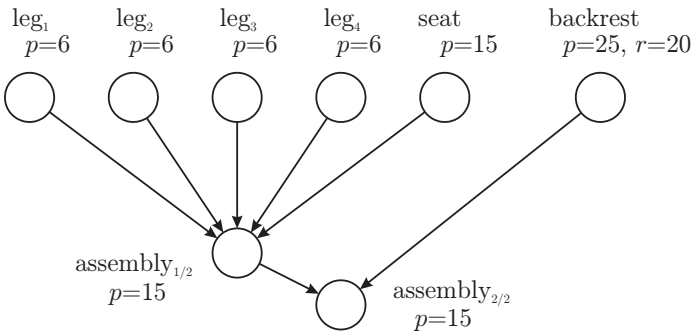


Fig. 5. Graph representing the chair manufacturing

Solution of the scheduling problem is shown in the following steps:

1. Create desired tasks.

```
>> T1 = task('leg1',6)
Task "leg1"
 Processing time: 6
 Release time:    0

>> T2 = task('leg2',6);
>> T3 = task('leg3',6);
>> T4 = task('leg4',6);
>> T5 = task('seat',6);
>> T6 = task('backrest',25,20);
>> T7 = task('assembly1/2',15);
>> T8 = task('assembly2/2',15);
```

2. Define precedence constraints by adjacency matrix `prec`. Matrix has size $n \times n$ where n is the number of tasks.

```
>> prec = [0 0 0 0 0 1 0 0;...
           0 0 0 0 0 1 0 0;...
           0 0 0 0 0 1 0 0;...
           0 0 0 0 0 1 0 0;...
           0 0 0 0 0 1 0 0;...
           0 0 0 0 0 0 0 1;...
           0 0 0 0 0 0 0 1;...
           0 0 0 0 0 0 0 0];
```

3. Create an object of taskset from recently defined objects.

```
>> T = taskset([T1 T2 T3 T4 T5 T6 T7 T8],prec)
Set of 8 tasks
 There are precedence constraints
```

4. Define solved problem.

```
>> p = problem('P|rj,prec|Cmax')
P|rj,prec|Cmax
```

5. Call the List Scheduling algorithm on taskset `T` where the scheduling problem is defined by object `p` and number of processors available for manufacturing is equal to 2. The algorithm use the Shortest Processing Time first (SPT) strategy.

```
>> TS = listsch(T,p,2,'SPT')
Set of 8 tasks
 There are precedence constraints
 There is schedule: List Scheduling
   Solving time: 0.1404s
```

6. Display the final schedule by standard plot function, see Fig. 6.

```
>> plot(TS)
```

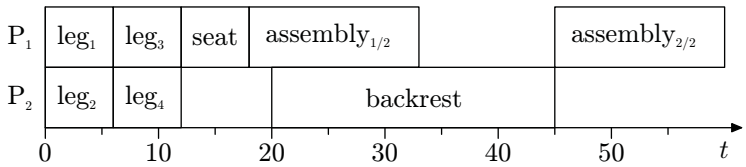


Fig. 6. The chair manufacturing schedule

3.2 Scheduling on Parallel Identical Processors

This section presents a SAT (satisfiability of boolean expression) based algorithm for the scheduling problem $P|prec|C_{max}$, i.e. the scheduling of tasks with precedence constraints on the set of parallel identical processors while minimizing the schedule makespan. The main idea is to formulate the scheduling problem in the form of CNF (conjunctive normal form) clauses (Crama & Hammer, 2006; Memik & Fallah, 2002).

In the case of the $P|prec|C_{max}$ problem, each CNF clause is a function of the boolean variables in the form x_{ijk}. If the task T_i is started at time unit j on the processor k then $x_{ijk} = true$, otherwise $x_{ijk} = false$. For each task T_i, where $i = 1 \ldots n$, there are $S \times R$ Boolean variables, where S denotes the maximum number of time units and R denotes the total number of processors.

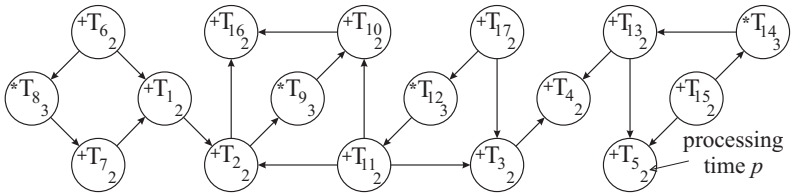


Fig. 7. Jaumann wave digital filter

```
>> procTime = [2,2,2,2,2,2,2,3,3,2,2,3,2,3,2,2,2];
>> prec = sparse([6,7,1,11,11,17,3,13,13,15,8,6,2, 9,11,12,17,14,15,2 ,10],...
                 [1,1,2, 2, 3, 3,4, 4, 5, 5,7,8,9,10,10,11,12,13,14,16,16],...
                 [1,1,1, 1, 1, 1,1, 1, 1, 1,1,1,1, 1, 1, 1, 1, 1, 1, 1, 1],17,17);
>> TS = taskset(procTime,prec);
>> TS = satsch(TS,problem("P|prec|Cmax"),2)
Set of 17 tasks
 There are precedence constraints
 There is schedule: SAT solver
   SUM solving time: 0.06s
   MAX solving time: 0.04s
   Number of iterations: 2
>> plot(TS)
```

Fig. 8. Solution of the scheduling problem $P|prec|C_{max}$ in the toolbox

The Boolean variables are constrained by the following three rules: 1. For each task, exactly one of the $S \times R$ variables has to be equal to *true*. 2. If there are precedence constraints such that T_u is the predecessor of T_v, then T_v cannot start before the execution of T_u is finished. 3. At any time unit, there is at most one task executed on a given processor. The rules result in a set of clauses in CNF generated by the algorithm in the toolbox that are consequently solved in a selected SAT solver.

The toolbox cooperates with the *zChaff* solver (Moskewicz et al., 2001) to decide whether the set of clauses is satisfiable. If it is, the schedule within S time units is feasible. An optimal schedule is found in an iterative manner. First, the List Scheduling algorithm is used to find the initial value of S. Then the algorithm iteratively decreases the value of S by one and tests the feasibility of the solution. The iterative algorithm finishes when the solution is not feasible.

An example of the $P|prec|C_{max}$ problem can be taken from the digital signal processing area. A typical scheduling problem is to optimize the speed of a computation loop, e.g constituting the Jaumann wave digital filter (de Groot et al., 1992). The goal is to minimize the computation time of the filter loop, shown as a directed acyclic graph in Fig. 7. The nodes in the graph represent the tasks (i.e. operations of the loop) and the edges represent the precedence constraints. The nodes are labeled by the operation type ("+" or "∗") and processing time p_i. The example in Fig. 7 considers two parallel identical processors, i.e. two general arithmetic units.

Fig. 8 shows the consecutive steps performed in the toolbox. The first step defines the set of the tasks with the precedence constraints for the scheduling algorithm `satsch`. The resulting schedule is displayed by the `plot` command. The optimal schedule is depicted in Fig. 9.

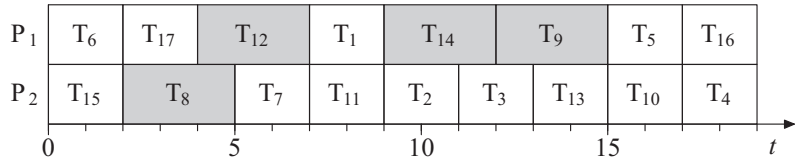


Fig. 9. The optimal schedule of the Jaumann filter

3.3 Cyclic Scheduling

The subsequent example has its application in digital signal processing (DSP) too but it uses a cyclic approach. *Cyclic scheduling* deals with a set of tasks (operations) that have to be performed an infinite number of times (Hanen & Munier, 1995). This approach is also applicable if the number of loop repetitions is large enough. If the execution of the operations belonging to different iterations can interleave, the schedule is called *overlapped*. An overlapped schedule can be more effective especially if processors are pipelined hardware units or precedence delays are considered. The *periodic schedule* is a schedule of one iteration that is repeated with a fixed time interval called a *period* (also called the *initiation interval*). The aim is then to find a periodic schedule with a minimum period (Hanen & Munier, 1995).

3.3.1 The Problem Outline

A DSP algorithm, used as an example, is a wave digital filter (LWDF) (Vesterbacka et al., 1994). Its computation loop, shown in Fig. 10(a), consists of five tasks. Their processing times are given by parameters of the used floating point arithmetic library for FPGA (Field-programmable gate array) circuits.

The operations in a computation loop can be considered as a set of n tasks $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ to be performed N times where N is usually very large. One execution of $\mathcal{T}$ labeled with the integer index $k \geq 1$ is called an *iteration*. The scheduling problem is to find a start time $s_i(k)$ of every occurrence T_i.

The data dependencies of this problem can be modeled by a directed graph $G(V, E)$. Each task (node in V) is characterized by the processing time p_i. Edge $(i, j) \in E$ from the node i to j is weighted by a couple of integer constants l_{ij} and h_{ij}. Length l_{ij} represents the minimal distance in clock cycles from the start time of the task T_i to the start time of T_j and is always greater than zero. Its value corresponds to the input-output latency of the used arithmetic

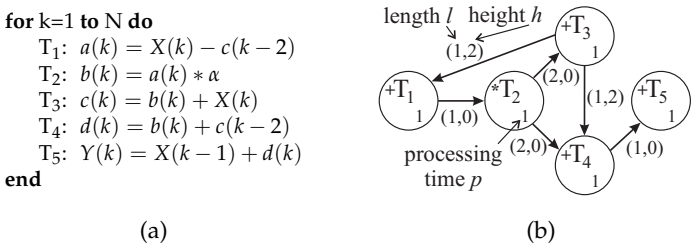


Fig. 10. Lattice wave digital filter (LWDF)

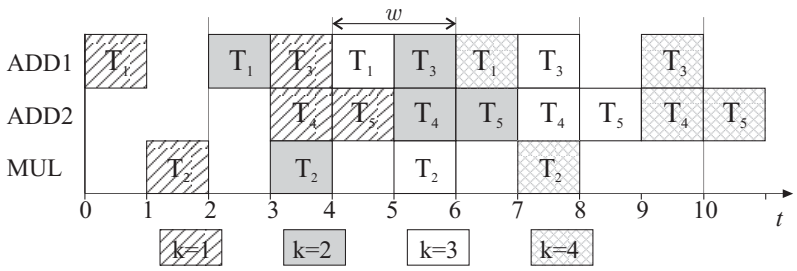


Fig. 11. An optimal schedule with period $w = 2$

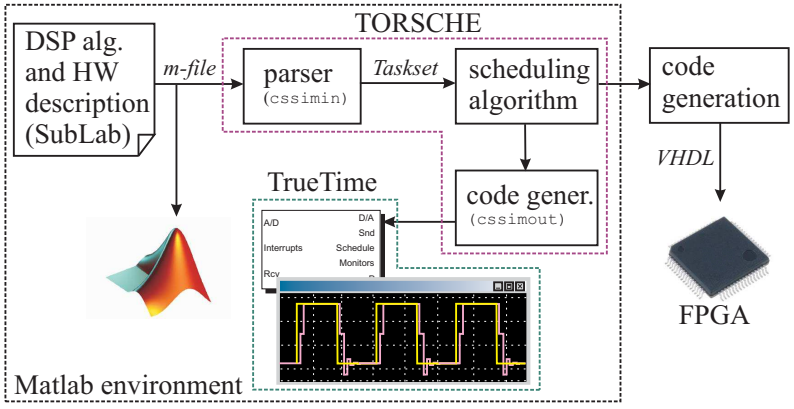


Fig. 12. Code to code generation

units. In our example in Fig. 10(b), the input-output latency of ADD (MUL) unit is 1 (2) clock cycles. On the other hand, the height h_{ij} specifies the shift of the iteration index (dependence distance) related to the data produced by T_i and read (consumed) by T_j.
Assuming a *periodic schedule* with the *period* w (i.e. the constant repetition time of each task), each edge $(i, j) \in E$ represents one precedence relation constraint $s_j - s_i \geq l_{ij} - w \cdot h_{ij}$, where s_i denotes the start time of task T_i in the first iteration. Fig. 10(b) shows the data dependence graph of the computation loop shown in Fig. 10(a). When the number of processors m is restricted, the cyclic scheduling problem becomes NP–complete (Hanen & Munier, 1995). An optimal solution of the example from Fig. 10 is shown in Fig. 11.

3.3.2 Solution in the Toolbox

In the toolbox, we formulate this scheduling problem as a problem of Integer Linear Programming (Šůcha & Hanzálek, 2009). In addition, the toolbox provides the possibility to simulate scheduled iterative loops in Matlab/Simulink using TrueTime tool (Andersson et al., 2005). The idea is depicted in Fig. 12. An instance of this scheduling problem (*Taskset* object) can be created manually but it can be generated automatically from a DSP algorithm and HW description in *SubLab*, a language compatible with Matlab. The main advantage of *SubLab* is that

```
function Y=lwdf(X)

%Arithmetic Units Declaration
struct('operator','+','number',2, ...
       'proctime',1,'latency',1);
struct('operator','*','number',1, ...
       'proctime',1,'latency',2);

struct('frequency',2000000);

%Variables Declaration
K = 10000;       alpha = 0.375;
a = zeros(1,K);  b = zeros(1,K);
c = zeros(1,K);  d = zeros(1,K);
Y = zeros(1,K);

%Iterative Algorithm
for k=3:K
    a(k) = X(k) - c(k-2);
    b(k) = a(k) * alpha;
    c(k) = b(k) + X(k);
    d(k) = b(k) + c(k-2);
    Y(k) = X(k-1) + d(k);
end
```

Fig. 13. LWDF algorithm in SubLab

```
>> [TS,m]=cssimin(dsvffile,schoptions);
>> prob=problem('CSCH');
>> TS=cycsch(TS, prob, m, schoptions);
>> plot(TS);
>> cssimout(TS,'simple_init.m','code.m');
```

Fig. 14. Solution of a cyclic scheduling problem in the toolbox

its code can be both transformed into the *Taskset* object and directly executed in the Matlab environment in order to check the iterative loops that were input (see Fig. 13).
Fig. 14 shows how the cyclic scheduling problem can be solved in the toolbox in three steps: input file parsing (function `cssimin`), cyclic scheduling (function `cycsch`) and True-Time code generation (function `cssimout`). The simulation is realized so that the scheduled code is time-exact executed by the *TrueTime Kernel* block which can be interconnected with other Matlab/Simulink blocks. It allows one to directly verify the behavior of the scheduled DSP algorithm interconnected with an appropriate model of the dynamic system.
Furthermore, the scheduling results can be used to generate parallel code for FPGAs in VHDL language. The concept is the same as for the simulation in TrueTime (see Fig. 12) but the output is a VHDL file which can be embedded into an FPGA design defining arithmetic units, interfaces, etc. FPGA code generation is not freely available in TORSCHE.

3.4 Minimum Cost Multi-commodity Flow Problem

Various optimization problems (e.g. routing) from the graph and network flow theory can be reformulated in terms of a minimum cost multi-commodity flow (MMCF) problem. The objective of the MMCF is to find the cheapest possible ways of sending a certain amounts of flows through the network. Therefore, TORSCHE includes a `multicommodityflow` function.

The MMCF problem is defined by a directed flow network graph $G(V, E)$, where the edge $(u, v) \in E$ from node $u \in V$ to node $v \in V$ has a capacity c_{uv} and a cost a_{uv}. There are ψ commodities $K_1, K_2, \ldots, K_\psi$ defined by $K_i = (source_i, sink_i, b_i)$ where $source_i$ and $sink_i$ stand for source and sink node of commodity i, and b_i is the volume of the demand. The flow of commodity i along the edge (u, v) is $f_i(u, v)$. The objective is to find an assignment of the flow $f_i(u, v)$ which minimizes the total cost $J = \sum_{\forall (u,v) \in E} \left(a_{uv} \cdot \sum_{i=1}^{\psi} f_i(u, v) \right)$ and satisfies the following constraints:

$$\begin{array}{rcll} \sum_{i=1}^{\psi} f_i(u, v) & \leq & c_{uv} & \forall (u, v) \in E, \\ \sum_{u \in V} f_i(u, w) & = & \sum_{v \in V} f_i(w, v) & w \in V \setminus \{source_i, sink_i\}, \\ & & & \forall i = 1 \ldots \psi, \\ \sum_{w \in V} f_i(source_i, w) & = & \sum_{w \in V} f_i(w, sink_i) = b_i & \forall i = 1 \ldots \psi. \end{array}$$

The function `multicommodityflow` solves MMCF problem by the transformation to the linear programming problem (Korte & Vygen, 2006).

3.4.1 Example of the Urban Traffic Scheduling

As an example, we show how to find the time schedule for light-controlled intersections in a urban traffic region in Prague by the TORSCHE toolbox.

The light controlled intersections are characterized by several parameters: the number of light phases, phase split, offset time and a list of streets from which the vehicles flow (Guberinić et al., 2008). The term phase means state of traffic lights on the intersection. The number of phases and the list of streets are partially given by the urban architecture of the intersection and partially by the intersection control strategy (i.e. one-way street, directional roadway marking). Both of these parameters are constant. On the other hand, the split and offset can be changed dynamically during a day. The *split* τ_{vj} defines the time interval of phase j for which the vehicle flow can go through the intersection v from one or more streets (Papageorgiou et al., 2003). The *offset* φ_{uv} is a certain time delay between phases of two successive intersections u and v. When the offset is zero, all lights in the region turn on and off at the same time. It is called the synchronized strategy. In the *green wave* strategy, the traffic light changes with time delay between the light phases of two successive intersections. As a result, signals switch as the green wave (Nagatani, 2007).

The goal is to find the offset φ_{uv} respecting the green wave strategy and the split τ_{vj} which minimizes the total cost J and considers a constant *control period* $P_v = \sum_{\forall j} \tau_{vj}$ of intersection v such that $P_1 = P_2 = \cdots = P$.

For the first step, a traffic region is modeled as an oriented graph $G(V, E)$. Nodes V of the graph represent the intersections and edges E represent the streets. See Fig. 15 where the *Graphedit* tool of TORSCHE is utilized to construct the graph. Sink and source nodes are drawn as rectangles. The edges include two parameters; the first one is cost a_{uv} and the second one is capacity c_{uv} of the street (u, v). The cost is given by the street length in meters. The capacity of the street is given by the number of lanes ℓ_{uv} in the street as $c_{uv} = \ell_{uv} \cdot W_{uv}/l$ where W_{uv} is a maximal allowed vehicle speed in the street in ms^{-1} and l is the *unit vehicle* length including distance between vehicles. Let us assume that, in our case, the speed is $W_{uv} = 13.8\,\mathrm{ms}^{-1}$ ($50\,\mathrm{km/h}$) and the unit vehicle length is $l = 5\,\mathrm{m}$, then the capacity of one lane street is $2.8\,\mathrm{s}^{-1}$. The final graph is exported from the *Graphedit* tool to the Matlab workspace as graph object `G`.

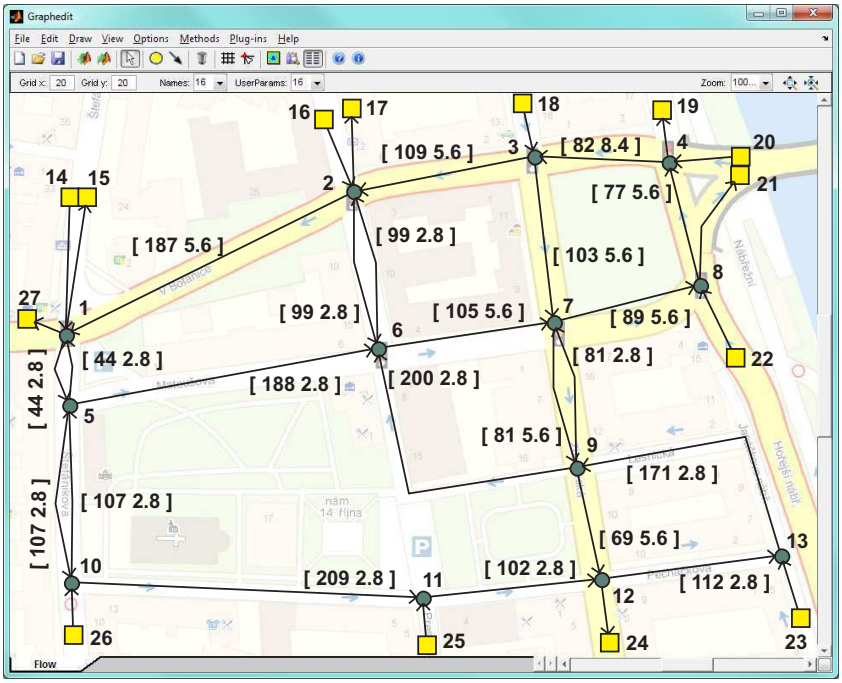


Fig. 15. Traffic region model

In the second step, the multicommodity flow method in the following form is called:

```
>> Gm = multicommodityflow(G,source,sink,b)
```

where the vectors `source`, `sink` and `b` define the required multi-commodity flow as is it described above. These variables are shown in Table 1. The graph `Gm` includes an assignment of optimal multi-commodity flow to the edges. We assume the drivers make their decision in a similar way. Table 2 shows a part of the assignment $f_i(u,v) : u = 6 \vee v = 6$. The complete result can be obtained from the *Graph* object by the command:

```
>> F = get(Gm,'edl')
```

3.4.2 Tasks Definition for Intersection

In the last step, the phase j split τ_{vj} and the offset φ_{uv} for each light controlled intersection $v \in V$ are found. Continuous vehicle flow from the street (u,v) over a given number of intersection phases can be formalized as one task T_{uv} from the scheduling theory point of view. The number of phases is given by the urban architecture of the intersection and by the intersection control strategy. The processing time p_{uv} of task T_{uv} is calculated by Algorithm 1 (Part of the Algorithm shows the solution of the consecutive steps in TORSCHE for intersection 6). This algorithm computes the processing time from the assignment of MMCF $f_i(u,v)$, from the control period P and from the precedence constraints of the tasks (defined by the intersection control strategy).

K_i	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8
$source_i$	14	14	14	14	16	16	16	16
$sink_i$	24	17	21	27	24	21	15	27
$b_i[10^{-3}s^{-1}]$	4.9	1.9	3.1	30.1	10.1	12.1	1.2	6.6
K_i	K_9	K_{10}	K_{11}	K_{12}	K_{13}	K_{14}	K_{15}	K_{16}
$source_i$	18	18	18	18	18	20	20	20
$sink_i$	24	17	21	15	27	19	24	17
$b_i[10^{-3}s^{-1}]$	32.7	3.3	9.3	2.1	11.3	8.9	13.4	8.1
K_i	K_{17}	K_{18}	K_{19}	K_{20}	K_{21}	K_{22}	K_{23}	K_{24}
$source_i$	20	20	22	22	22	22	23	23
$sink_i$	15	27	19	21	15	27	24	17
$b_i[10^{-3}s^{-1}]$	7.9	41.7	11.7	88.5	4.7	25.1	8.6	4.3
K_i	K_{25}	K_{26}	K_{27}	K_{28}	K_{29}	K_{30}	K_{31}	K_{32}
$source_i$	23	23	25	25	26	26	26	26
$sink_i$	21	27	24	21	24	21	15	27
$b_i[10^{-3}s^{-1}]$	6.5	3.6	9.6	0.4	15.9	1.9	5.7	6.5

Table 1. Required traffic region multi-commodity flow instances

Algorithm 1 Processing time computation

1. Create tasks T_{uv}, each with temporary processing time $p'_{uv} = \sum_{i=1}^{\psi} f_i(u,v)$.

```
>> T56 = task('T(5,6)', 0.0069)
>> T26 = task('T(2,6)', 0.0222);
>> T96 = task('T(9,6)',0.0079);
```

2. Group the tasks into a *taskset* and add precedence constrains.

```
>> prec6 = [0 1 1; 0 0 0; 0 0 0];
>> TS6 = taskset([T56 T26 T96],prec6);
```

3. Compute a length of critical path CP_v by the *asap* (as soon as possible) function.

```
>> TS6.asap;
>> asapStart = asap(TS6,'asap');
>> CP6 = max(asapStart + TS6.ProcTime)
CP6 =
    0.0291
```

4. From the length of the critical path and control period P we obtain processing time p_{uv} as a linear proportion of flow: $p_{uv} = p'_{uv} \cdot P/CP_v$.

```
>> P = 90;
>> TS6.ProcTime = TS6.ProcTime * P / CP6;
```

	K_i	K_2	K_3	K_5	K_6	K_{24}	K_{26}	K_{30}	
u	v	$f_i(u,v)$ $[10^{-3}s^{-1}]$							$\sum_{i=1}^{\psi} f_i(u,v)$
2	6	0	0	10.1	12.1	0	0	0	22.2
5	6	1.9	3.1	0	0	0	0	1.9	6.9
9	6	0	0	0	0	4.3	3.6	0	7.9
6	2	1.9	0	0	0	4.3	3.6	0	9.8
6	7	0	3.1	10.1	12.1	0	0	1.9	27.2

Table 2. Multi-commodity flow assignment

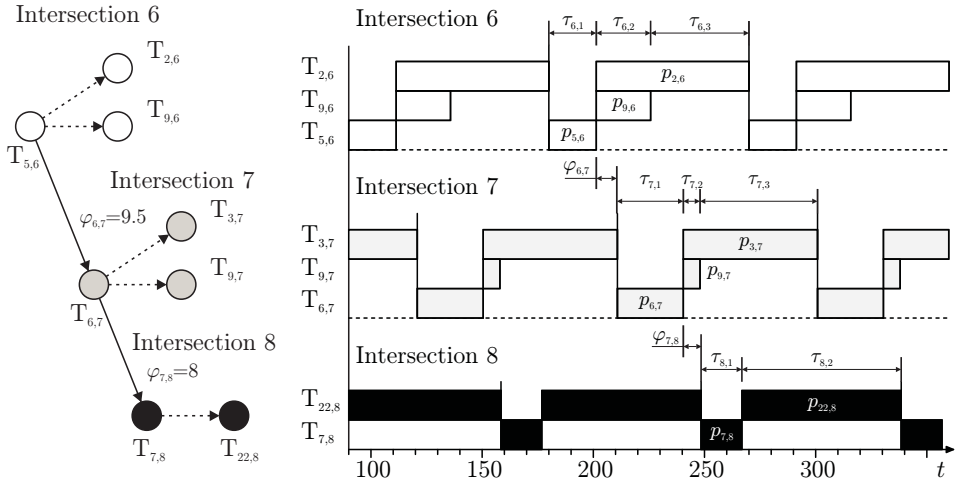
3.4.3 Scheduling with Communication Delay

The intersection phase offset and split are computed for the green wave strategy. The green wave strategy, specified by the engineering skills, extends the tasks precedence constraints by the relationships between successive intersection tasks. Each of those relationships defines the offset φ_{uv} as a time, which a vehicle needs to pass from intersection u to intersection v. The φ_{uv} is given by the street length a_{uv} and vehicle speed W_{uv} as $\varphi_{uv} = a_{uv}/W_{uv}$.
The split can be found by an algorithm for scheduling with a communication delay (Chrétienne et al., 1995). The *scheduling with communication delay* problem extends the precedence constraints in the classical scheduling by the communication delay between dependent tasks assigned to distinct processors. In our case the communication delay is equal to the offset φ_{uv}. Let `D` be a matrix of communication delays, where the elements are φ_{uv} in the case that the offset between intersections u and v is considered, and zero otherwise. We can classify our instances as tasks with precedence constraints in an out-tree form, communication delays, unlimited number of processors and no duplication of tasks. In Graham and Błażewicz notation it can be denoted as $P_\infty|out\text{-}tree, c_{jk}|C_{max}$. This problem can be solved in $\mathcal{O}(n)$ time by the allgorithm presented in (Chrétienne, 1989) which is implemented in the TORSCHE toolbox as a function `chretienne`.
Fig. 16 shows the problem solution in the toolbox for three intersections (6, 7 and 8).

```
>> T56 = task('T(5,6)', 21.3);
>> T26 = task('T(2,6)', 68.7);
>> T96 = task('T(9,6)', 24.4);
>> prec6 = [0 1 1; 0 0 0; 0 0 0];
>> TS6 = taskset([T56 T26 T96],prec6);
>> T67 = task('T(6,7)', 29.6);
>> T37 = task('T(3,7)', 60.4);
>> T97 = task('T(9,7)',  7.5);
>> prec7 = [0 1 1; 0 0 0; 0 0 0];
>> TS7 = taskset([T67 T37 T97],prec7);
>> T78 = task('T(7,8)',  18.4);
>> T228 = task('T(22,8)', 71.6);
>> prec8 = [0 1; 0 0];
>> TS8 = taskset([T78 T228],prec8);
>> TSall = [TS6 TS7 TS8];
>> TSall.Prec(1,4) = 1;
>> TSall.Prec(4,7) = 1;
>> D = zeros(size(TSall.Prec));
>> D(1,4) = 9.5;
>> D(4,7) = 8;
>> prob = ...
   problem('Pinf|prec,out-tree,cjk|Cmax');
>> TSall = chretienne(TSall,p,Inf,D);
>> plot(TSall);
```

Fig. 16. Solution of the scheduling problem in the toolbox

First, the taskset object `TSall` with eight tasks corresponding to the intersection control is defined. The tasks and precedence constraints among them are shown in Fig. 17(a). The precedence constraints given by the green-wave strategy are drawn as solid lines. Consequently, matrix `D` and the notation of the problem `prob` is defined. Finally, the scheduling problem



(a) Precedence constraints (b) Gantt chart for vehicle flow including split and offset mark

Fig. 17. The intersections (6, 7 and 8) control

is solved by the algorithm `chretienne` and the resulting Gantt chart is shown in Fig. 17(b). The figure shows the tasks for the three considered intersections including the processing time p_{uv}, split τ_{vj} and offset φ_{uv}. The split is given by the processing time of the scheduled tasks. The tasks are repeated with period P.

3.5 Visualization of the Scheduling Results

In the toolbox, there is a possibility to simulate or visualize the acquired scheduling results in the Matlab/Simulink environment through the use of the Matlab Virtual Reality toolbox, which is a standard part of Matlab. TORSCHE allows users to create their own project with 3-D animation in the Virtual Reality toolbox and interconnect it with a schedule obtained in TORSCHE. Consequently, both simulation and visualization are realized in Matlab/Simulink and in case of visualization, it is possible to capture any frame or a stream video file of the animation.

The hoist scheduling problem is chosen as an example for the visualization. The hoist scheduling problem (Manier & Bloch, 2003) deals with the problem how to schedule the hoist movements to perform a material handling between several tanks with elecrolyte solution, where the material is processed. The objective of this problem is to find a schedule which maximizes the processing throughput. The hoist scheduling problem can be solved by the `singlehoist` algorithm that is available in the TORSCHE toolbox. The problem is represented so that the tasks correspond with the movements of the hoist (load/unload station → Bath 1, Bath 1 → Bath 2, Bath 2 → Bath 3, Bath 3 → load/unload station).

The problem solution in the toolbox is shown in Fig. 18(a). The interconnection between the tasks and the project with 3-D animation is realized by a user defined text-file. Fig. 18(b) shows a fragment of the text-file containing the code that describes the movement of the hoist corresponding to the first task. The interconnection is performed in the function `adduserparam`. This function parses the code of the input-text file and

```
>> a = [0 70 70 30];
>> b = [0 100 200 75];
>> C = toeplitz([0 15 20 25]);
>> d = [36 36 36 51];
>> T = taskset(d);
>> T.TSUserParam.SetupTime    = C;
>> T.TSUserParam.minDistance = a;
>> T.TSUserParam.maxDistance = b;
>> TS = singlehoist(T);
>> adduserparam(TS,'onehoist.txt');
>> name  = 'onehoist.wrl';
>> ports = visiscontrolports('Output',
           'HoistArm',3,...
>> VRin  = vrports('Hoist','translation',...
>> taskset2simulink(name, TS, ports, VRin,
                    500, []);
```

(a) Solution in the toolbox

```
%File name: 'onehoist.txt'
task1
repeat 0:1:4
HoistArm(1) = HoistArm(1) + 1;
repeat 0:1:7
HoistArm(2) = HoistArm(2) - 0.5;
...
endparam

task2
repeat 0:1:5
HoistArm(1) = HoistArm(1) - 1;
repeat 0:1:9
HoistArm(2) = HoistArm(2) + 1;
...
endparam
```

(b) Visualization description (file: onehoist.txt)

Fig. 18. Matlab code for hoist scheduling visualization

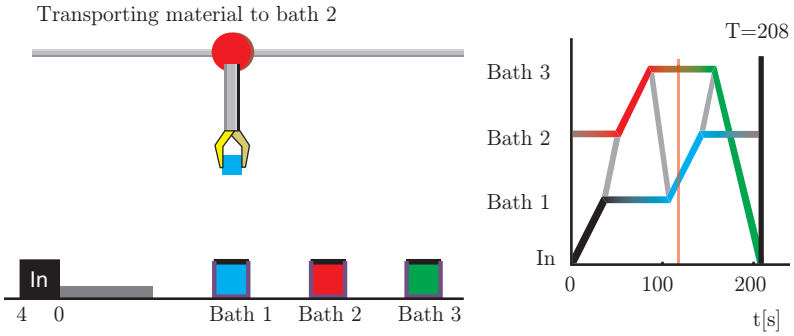


Fig. 19. Visualization of the hoist scheduling problem

assigns an appropriate code of visualization to each task. The visualization is subsequently performed by executing an automatically generated Matlab/Simulink file. The resulting problem visualization is shown in Fig. 19, and a video file can be found on `http://rtime.felk.cvut.cz/scheduling-toolbox/video`.

4. Conclusions

This chapter presents the TORSCHE Scheduling Toolbox for Matlab covering: scheduling on monoprocessor/dedicated processors/parallel processors, open shop/flow shop/job shop scheduling, cyclic scheduling and real-time scheduling. The toolbox includes scheduling algorithms that have been used for various applications as scheduling of Digital Signal Processing algorithms on a hardware architecture with pipelined arithmetic units, scheduling the movements of hoists in a manufacturing environment and scheduling of light controlled intersections in urban traffic. Moreover, the toolbox already has several real applications. It has been used for the development of a new method for re-configuration of the tasks or a process in an embedded avionics application (Muniyappa, 2009). Simulations in TORSCHE

also helped to develop a method optimizing the jitter of tasks in a real-time system (Liu et al., 2009). Recently, TORSCHE has become a part of a textbook for courses in scheduling "Scheduling: Theory, Algorithms, and Systems" written by M. Pinedo (Pinedo, 2008). The actual version of the toolbox with documentation and screencasts is freely available at `http://rtime.felk.cvut.cz/scheduling-toolbox/`.

5. References

Andersson, M., Henriksson, D. & Cervin, A. (2005). *TrueTime 1.3–Reference Manual*, Department of Automatic Control, Lund University, Sweden, Lund University, Sweden. `http://www.control.lth.se/truetime/`.

Andresen, M., Bräsel, H., Engelhardt, F. & Werner, F. (2003). *LiSA - A Library of Scheduling Algortihms*, Otto-von-Guericke-Universität Magdeburg. `http://lisa.math.uni-magdeburg.de/`.

Blazewicz, J., Lenstra, J. & Kan, A. R. (1983). Scheduling subject to resource constraints. Classification and complexity, *Discrete Applied Mathematics* **5**(5): 11–24.

Chrétienne, P. (1989). A polynomial algorithm to optimally schedule tasks on a virtual distributed system under tree-like precedence constraints, *European Journal of Operational Research* **43**(43): 225–230.

Chrétienne, P., Coffman, E. G., Lenstraand, J. K. & Liu, Z. (eds) (1995). *Scheduling theory and its applications*, John Wiley & Sons Ltd, Baffins Lane, Chichester, West Sussex PO19 1UD, England.

Crama, Y. & Hammer, P. L. (2006). Boolean functions: Theory, algorithms and applications. `http://www.rogp.hec.ulg.ac.be/Crama/Publications/BookPage.html`.

de Groot, S. H., Gerez, S. & Herrmann, O. (1992). Range-chart-guided iterative data-flow graph scheduling, *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on* **39**: 351–364.

Gonzalez, M. et al. (2008). MAST (Modeling and Analysis Suite for Real-Time Applications), `http://mast.unican.es/`.

Guberinić, S., Šenborn, G. & Lazić, B. (2008). *Optimal Traffic Control: Urban Intersections*, CRC Press, 6000 Broken Sound Parkway NW, Suite 300, Boca Raton.

Hanen, C. & Munier, A. (1995). A study of the cyclic scheduling problem on parallel processors, *Discrete Applied Mathematics* **57**: 167–192.

ILOG (2009). *ILOG CP Optimizer*, IBM Corporation, ILOG Europe, 9 rue de Verdun, BP 85, 94253 Gentilly Cedex. `http://www.ilog.com/products/cpoptimizer/`.

Korte, B. H. & Vygen, J. (2006). *Combinatorial Optimization: Theory and Algorithms*, third edn, Springer-Verlag, Berlin, Heidelberg.

Kutil, M., Šůcha, P., Sojka, M. & Hanzálek, Z. (2007). *TORSCHE Scheduling Toolbox for Matlab: User's Guide*, Centre for Applied Cybernetics, Department of Control Engineering, Czech Technical University in Prague. `http://rtime.felk.cvut.cz/scheduling-toolbox/`.

Leung, J. Y.-T. (2004). *Handbook of Scheduling*, Chapman & Hall/CRC.

Liu, Z., Zhao, H., Li, P. & Wang, J. (2009). An optimization model for io jitter in device-level rtos, *ITNG '09: Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*, IEEE Computer Society, Washington, DC, USA, pp. 1528–1533.

Manier, M. A. & Bloch, C. (2003). A classification for hoist scheduling problems, *International Journal of Flexible Manufacturing Systems* **15**(1): 37–55.

Memik, S. O. & Fallah, F. (2002). Accelerated SAT-based scheduling of control/data flow graphs, *ICCD '02: Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, IEEE Computer Society, Washington, DC, USA, p. 395.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L. & Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver, *Proceedings of the 38th Design Automation Conference (DAC'01)*, ACM, pp. 530–535.

Muniyappa, A. C. (2009). *Computer Safety, Reliability, and Security*, Springer, Berlin Heidelberg.

Nagatani, T. (2007). Vehicular traffic through a sequence of green-wave lights, *Physica A: Statistical Mechanics and its Applications* **380**: 503 – 511.

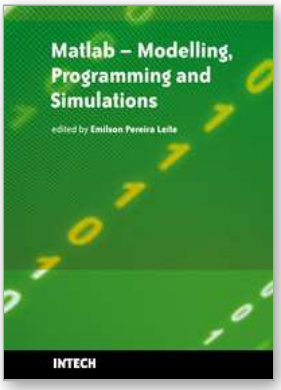
Papageorgiou, M., Diakaki, C., Dinopoulou, V., Kotsialos, A. & Wang, Y. (2003). Review of Road Traffic Control Strategies, *PROCEEDINGS - IEEE* **91**: 2043–2067.

Pinedo, M. (2008). *Scheduling: Theory, Algorithms, and Systems*, third edn, Springer, 233 Spring Street, NewYork, NY 10013, USA.

Pinedo, M. et al. (2002). *LEKIN® - Flexible Job-Shop Scheduling System*, Stern School of Business, NYU, New York University, Leonard N. Stern School of Business New York, NY. `http://www.stern.nyu.edu/om/software/lekin/`.

Šůcha, P. & Hanzálek, Z. (2009). A cyclic scheduling problem with an undetermined number of parallel identical processors, *Computational Optimization and Applications* . article in press.

Vesterbacka, M., Palmkvist, K., Sandberg, P. & Wanhammar, L. (1994). Implementation of fast dsp algorithms using bit-serial arithmetic, *National Conference on Electronic Design Automation, Stockholm*.



Matlab - Modelling, Programming and Simulations

Edited by Emilson Pereira Leite

ISBN 978-953-307-125-1
Hard cover, 426 pages
Publisher Sciyo
Published online 05, October, 2010
Published in print edition October, 2010

This book is a collection of 19 excellent works presenting different applications of several MATLAB tools that can be used for educational, scientific and engineering purposes. Chapters include tips and tricks for programming and developing Graphical User Interfaces (GUIs), power system analysis, control systems design, system modelling and simulations, parallel processing, optimization, signal and image processing, finite different solutions, geosciences and portfolio insurance. Thus, readers from a range of professional fields will benefit from its content.

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821