Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Tomáš Skalický

Interactive Gantt Charts

Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Doc. RNDr. Roman Barták, Ph.D.

Field of study: Computer Science, Programming

2008

Acknowledgements

I thank Doc. RNDr. Roman Barták, Ph.D. for a direct and patient leading during working on the thesis.

I hereby certify that I wrote the thesis myself, using only the referenced sources. I agree with lending the thesis.

Prague, December 8, 2008 Tomáš Skalický

Content

Název práce: Interaktivní Ganttovy diagramy

Autor: Tomáš Skalický

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Doc. RNDr. Roman Barták, Ph.D.

e-mail vedoucího: bartak@ktiml.mff.cuni.cz

Abstrakt: Tato práce se zabývá problematikou interaktivního rozvrhování v Ganttových diagramech. Cílem práce je navrhnout a implementovat algoritmus, jež řeší nekonzistence v rozvrhu. Nejprve jsou rozebrány existující přístupy. Hlavním přínosem této práce je algoritmus, které řeší nekonzistence v rozvrhu. Je zde navržen a je zde také proveden důkaz jeho korektnosti a konečnosti. Následuje demonstrování jeho použitelnost. K tomuto účelu je použit program GanttViewer, jež je součástí práce. Na závěr je tento program podrobněji popsán.

Klíčová slova: interaktivní, přerozvrhování, rozvrh, Ganttův diagram

Title: Interactive Gantt Charts

Author: Tomáš Skalický

Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Doc. RNDr. Roman Barták, Ph.D.

Supervisor's e-mail address: bartak@ktiml.mff.cuni.cz

Abstract: This work deals with problems of interactive rescheduling in Gantt charts. The goal of the work is to design and implement an algorithm which resolves the inconsistencies in a schedule. First existing approaches are studied. Algorithm for solving our inconsistencies in a schedule is the most important contribution of this work. It is introduced there and there is also it's proof of correctness and finiteness. Demonstration of applicability of the algorithm follows. For this reason, program GanttViewer which is a part of this work is used. Finally, this program is described in detail.

Keywords: interactive, rescheduling, schedule, Gantt chart

Introduction

Constraint programming (CP) is a technology for solving combinatorial problems especially in such domains as artificial intelligence, computer science and others. In CP, relationships between variables are used for a description of problem. Problems, we can successfully solve with CP, are for example planning and scheduling.

Scheduling is a process of deciding how to allocate time and resources to perform some tasks. Constraints are relationships between tasks (e.g. task A and task B must start at the same time), resource capacities, task priorities etc. Interactive scheduling is an user-friendly type of scheduling.

In interactive scheduling, user can make changes while a schedule is being created. It means that if he wants to change something during running a scheduling algorithm, he stops the algorithm, make changes and finally he runs the algorithm again. There are various types of changes. User can add or remove a dependence (relationship between two tasks), change parameters (such as task duration) or add or remove a task or a resource. In a difficult case, user can help the algorithm by making changes to get a solution. For taking advantage of the interactive scheduling, we can use an interactive Gantt chart.

Gantt chart is a type of a horizontal bar chart that illustrates a project schedule. In the chart, there is time on x (horizontal) axis and tasks on y (vertical) axis. Thus, tasks are represented by bars. Some Gantt charts also show dependencies between the tasks.

The aim of this thesis is to design and demonstrate an interactive scheduling algorithm within a Gantt chart. This algorithm would reschedule user's schedule to a feasible one (explained later in section 1.2) which would be similar to the user's schedule as much as possible.

The thesis is divided into four chapters. In the first chapter, we introduce the notation, make the fundamental terms clear and formalize the problem. In the second chapter, existing approaches are described. In the following chapter, design of the interactive scheduling algorithm is described. There is also an implementation in pseudo code and a proof of correctness and finiteness of the algorithm is given there. At the end, example run of the proposed algorithm is demonstrated and applicability of the algorithm is presented.

Chapter 1

Problem formalization

In the first part of this chapter, we introduce the basic notation. In the second part, we explain fundamental terms and in the last part, we formalize the problem of interative scheduling.

1.1 Notation

For tasks *A*, *B* and resource *R*, we introduce the following notation:

s_A	... start time of *A*
d_A	... duration of *A*
$A \rightarrow B$	... *A* is predecessor of *B*, *B* is a successor of *A* (explained later in part 1.2)
lp(A)	... the latest predecessor of *A* (predecessor C of *A* which has the greatest ($s_C + d_C$))
A(R)	... *A* is performed by *R*

1.2 Fundamental terms

Dependence is a temporal relationship between two tasks. In particular, dependence $A \rightarrow B$ means, that $s_B >= s_A + d_A$ is necessary to be valid. Task *A* is called a **predecessor** of task *B* and task *B* is called a **successor** of task *A*.

Constraint is a dependence or a relationship between two tasks *A*, *B* performed by the same resource R. For these tasks *A*, *B, A(R)*, *B(R)*, ($s_A + d_A <= s_B$) v ($s_A + d_A <= s_B$) is necessary to be valid. If constraint is broken, it is called **resource conflict**.

One task can be performed by **more than one** resource, but all resources per task are known in advance.

Schedule is an allocation time to all tasks. Then **feasible schedule** is a schedule that satisfies the following two constraints:

constraint 1: for every tasks *A*, *B* such that $A \rightarrow B$: $s_B >= s_A + d_A$ must be valid,

constraint 2: for each resource R and every tasks *A*, *B* such that *A(R)*, *B(R)*, $s_A <= s_B$: $s_A + d_A <= s_B$ must be valid.

Constraint 1 says, that a task does not start until all its predecessors have finished, and constraint 2 says, that a resource can perform at most one task at the time.

Below, there are terms with less formal definitions.

Schedule is **compact** if there are no unnecessary free spaces between predecessors and successors.

Modification of start time of a task in a schedule is **local** if difference between the old start time and the new one is small in comparison with time the schedule is spread over.

1.3 Formalization of a problem

Problem, we are solving in this thesis, is to design an algorithm which reschedules user's schedule in such a way, that the schedule is feasible.

Schedule has special **properties**:

- start and duration of a task are positive integers or 0,
- schedule can be spread out to the future.

We have the following **requirements** for the algorithm:

- it must be correct and finite,
- only a start time of a task can be modified,
- modifications are local; schedule is only rescheduled, not created from the beginning,
- schedule is as compact as possible.

Chapter 2

Existing approaches

In this chapter, existing approaches to rescheduling are described. There are also reasons why they are or why they are not appropriate for solving the problem.

2.1 Iterative Flattening Search

Iterative flattening search (IFS), introduced in [1] and [2], is an iterative improvement heuristic schema for makespan minimization in scheduling problems.

Before describing the IFS algorithm it is necessary to introduce the model on which the IFS schema is based. In this model, a schedule *schedule* is represented as a directed graph $G_{schedule}(T, E)$. T is the set of tasks, plus a fictitious t_{source} task which occurs before all other tasks and a fictitious t_{sink} task which occurs after all other tasks. E is the set of constraints defined between tasks in T. The set E consists of two subsets, $E = E_{orig} \cup E_{post}$, where E_{orig} is the set of precedence constraints originating from the problem definition and E_{post} is the set of precedence constraints posted to resolve resource conflicts. In general the directed graph $G_{schedule}(T, E)$ represents a set of temporal schedules. The set E_{post} is added in order to guarantee that at least one of those temporal schedules satisfies constraint 2. The algorithm iterates the following two steps:

Relaxation step: the first step; a feasible schedule is relaxed into a schedule which satisfies constraint 1, but does not need to satisfy constraint 2. Some precedence constraints are removed from E_{post}.

Flattening step: the second step; new precedence constraints are added to E_{post} to get a feasible schedule.

The above two steps are executed until a better (according to function *evaluate*) solution is found or a maximal number of iterations is executed.

```
function IFS(schedule, $p_r$, maxFail, maxRelaxes)
    $schedule_{best}$ = schedule
    counter = 0
    while counter =< maxFail do
        Relax(schedule, $p_r$, maxRelaxes)
        schedule = Flatten(schedule)
        if evaluate(schedule) < evaluate($schedule_{best}$) then
            $schedule_{best}$ = schedule
            counter = 0
        else
            counter = counter + 1
        end
    end
    return $schedule_{best}$
end.
```

The IFS (from [1] and [2]) general schema

```
procedure Relax(schedule, $p_r$, maxRelaxes)
    for 1 to maxRelaxes do
        for each $(t_i, t_j) \in$ CP(schedule) $\cap E_{post}$ do
            if random(0, 1) < $p_r$ then
                schedule = schedule \ $(t_i, t_j)$
end.
```

The Relax procedure

The IFS procedure takes four elements as input: (1) an initial schedule *schedule*; (2) a value $p_r \in [0, 1]$ designating the percentage of precedence constraints $pc_i \in E_{post}$ on the critical path (explained below) to be removed; (3) a positive integer *maxFail* which specifies the maximum number of non-makespan-improving moves that the algorithm will tolerate before terminating; and (4) a positive integer *maxRelaxes* which specifies the maximum number of relax iterations to be performed in the relaxation step. Let's go back to the IFS algorithm, after initialization, *schedule* is repeatedly modified within the while loop by the application of the Relax and Flatten procedures. In the case that a better schedule is found, the new schedule is stored in $schedule_{best}$ and the *counter* is reset to 0. Otherwise, if no improvement is found in *maxFail* iterations, the algorithm terminates and returns the best found schedule.

Relaxation step is based on the concept of *critical path*. A *path* in $G_{schedule}(T, E)$ is a sequence of tasks $t_1, t_2, ..., t_k$, where $(t_i, t_{i+1}) \in E$ with $i = 1, 2, ..., (k - 1)$. The length of a path (calculated by *evaluate* function) is the sum of the tasks' duration times and a *critical path* is a path from t_{source} to t_{sink} which determines the schedule's makespan. Therefore, any improvement of the length of the critical path requires changes of constraints situated on the *critical path*. Thus, the relaxation step retracts some members of E_{post} on the critical path. Precedence constraints are randomly selected from the current schedule.

Flattening step consists itself of two steps. The first step is *to construct an infinite capacity schedule*. In this schedule, dependencies are modeled and satisfied, but the resource constraints are ignored. The second step consists in *leveling resource*. Resource constraints are modeled. Detected resource conflicts are then resolved by iteratively adding precedence constraints between pairs of competing tasks. For further details on the flattening procedure the reader should refer to the original papers.

The priority number one of the algorithm is to shorten the length of the critical path, not to make local changes. So that, the final schedule can be totally different than the input one. Therefore, the IFS algorithm cannot be used to reschedule user's schedule.

2.2 Iterative Forward Search

Iterative forward search (IFS), introduced in [3], works iteratively. It uses two basic data structures: a set of tasks which are not scheduled (*unscheduled*) and a partial feasible schedule (*sch*), i.e. there are scheduled all tasks except those which are in *unscheduled*. At the beginning, the schedule is empty, all tasks are in the set of unscheduled tasks. Then, in each iteration, the algorithm has an effort to improve the schedule (explained below). The algorithm does not finish until all tasks are scheduled or a number of iterations reaches a limit (*maxIter*).

Users can pause the algorithm after an iteration, do some modifications (e.g. add a new task) and resume the algorithm.

Each iteration has three steps. First of all, all unscheduled tasks are evaluated and the algorithm takes the worst one (explained below). In the second step, all resource time where the selected task can be placed are evaluated and the best location (explained below) is selected. Finally, the selected task is placed on the selected location. However, this newly scheduled task can be in conflict with other scheduled tasks. These conflicting tasks are removed from the schedule and they are inserted into the set of unscheduled tasks. Hence, the schedule is partially feasible at the end of each iteration.

```
procedure IFS2(unscheduled, sch, maxIter)
    counter = 0
    while counter < maxIter and unscheduled not empty and none user
    interruption do
        task = task from unscheduled
        unscheduled = unscheduled – task
        location = the best location in schedule where task can be placed
        task is placed on location
        unscheduled = unscheduled + tasks removed from schedule because of
        task
        counter = counter + 1
    end
    return schedule
end.
```

The IFS (from [3]) general schema

In the first and the second steps, there are used heuristics. Both of them are implemented as a weighted sum of several values such as: How many times has the task been removed yet? How many dependences are formed by the task? (both are used for selecting the task) How many scheduled tasks will be in conflict with the selected task if we selected that location? (used for selecting location) etc.

In the third step of the IFS algorithm, all tasks which are predecessors or successors of the selected (the worst) task have to be removed from a schedule. Predecessors and successors of these removed tasks are also removed and this is done iteratively until we get a partially feasible schedule.

In the following iterations the algorithm schedule these tasks, but new locations of them can be totally different. It means that changes are not local which is our requirement to a rescheduling algorithm. Therefore, the IFS algorithm cannot be used to reschedule user's schedule.

Chapter 3

Rescheduling algorithm

Process

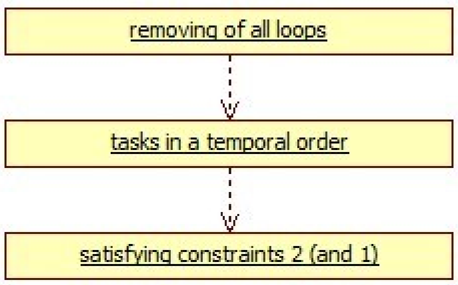
In the rescheduling algorithm, we use a graph $G_{schedule}(T, E)$ which represents a schedule. Tasks, which are in T, are vertices of the graph and properties "*be predecessor (successor) of*", which are in E, are oriented edges of the graph, i.e. for tasks $t_i, t_j \in T$ such that $t_i \rightarrow t_j$, $(t_i, t_j) \in E$.

The algorithm is divided into three phases processed in a sequence. It is not possible to start with the next phase until the previous one finished.

To satisfy constraint 1 (see section 1.2), which is about dependencies, we need to be able to organize tasks (vertices of the graph $G_{schedule}$) in a temporal order. When we remove loops which are formed by dependencies, we are able to do that. Thus, the first phase is **removing of all loops** in the graph. For this purpose, we use an algorithm which is based on finding vertices, which have no predecessors or no successors. This algorithm works iteratively and in each iteration, it removes dependencies selected by a user.

In the following two phases, we use a fact, that we can spread out a schedule over more time to the future. At first, we organize **tasks in a temporal order**, i.e. the schedule satisfies constraint 1. While constraint 1 is violated, an algorithm of this phase runs. In each iteration, the algorithm takes a pair of tasks A, B which breaks the constraint $A \rightarrow B$ and orders the tasks in such a way, that task A finishes before task B starts. By using the algorithm, the schedule can be spread out (to the future).

In the following phase and the last phase at all, a schedule is modified in such a way, that the schedule **satisfies constraint 2**, which is about resources, **and keeps validity of constraint 1 as well**. Algorithm of this phase iteratively checks whether tasks satisfy constraint 2. If it is true for all of them, the algorithm finishes and the schedule is feasible. If constraint 2 is broken, the algorithm shifts a pair of tasks, which breaks the constraint, in such a way, that the tasks do not overlap anymore. By this operation, some constraint 1 can be broken. Therefore, all tasks (successors), which breaks constraint 1, are shifted to the right. After that, we have a schedule which satisfy constraint 1 again.



graph 3.1: the phases of the rescheduling algorithm and an order of them

Interactivity of the rescheduling algorithm is realized in tree phases. First of all, the user stops the algorithm. Then, he makes changes in the schedule and finally, he starts the algorithm from the beginning.

3.1 Removing of all loops (RmL)

For removing of all loops of graph $G_{schedule}$ we use an iterative algorithm, which runs until there is a vertex in the graph. Each iteration consists of three steps:

1) iterative removing of vertices, which have no predecessors or no successors,
2) finding a loop,
3) removing of edges which belongs to the found loop and which are selected by an user. The user can select at least one or more edges.

Step 1: In this step, all vertices which have no predecessors or no successors are removed. If any vertex loses all predecessors or all successors by removing of those vertices, it is removed as well. This is done iteratively until we obtain a graph where each task has a predecessor and a successor, or the graph is empty.

Step 2: Now in graph $G_{schedule}$, there are only vertices which have at least one predecessor and at least one successor. If there is no vertex in the graph, the algorithm finishes. Otherwise, a loop must exists in the graph. In order to find the loop, we take any vertex and we go to one of it's successors. From this successor, we go to another successor and so on until we reach a vertex (v_i), we went through it before. The path between the first and the second occurrence of vertex v_i is a loop that we looked for.

Step 3: User selects at least one edge (dependence), which is a part of the found loop. Then, selected edges are removed to break the loop.

```
preCounts is initialized
sucCounts is initialized

while preCounts and sucCounts are not empty do
  STEP 1
  while the list of vertices on 0th position (the 0th list) of preCounts is not empty do
      u = the first vertex in the 0th list of preCounts
      u is removed from preCounts and from sucCounts
      for each successor of u (= $suc_u$) do
          u is removed from predecessors of $suc_u$
          $suc_u$ moves from the ith list (current position) of preCounts to the ($i$ – 1)th
          list
      end
  end
  while the 0th list of sucCounts is not empty do
      v = the first vertex in the 0th list of sucCounts
      v is removed from preCounts and from sucCounts
      for each predecessor of v (= $pre_v$) do
          v is removed from successors of $pre_v$
          $pre_v$ moves from the jth list (current position) of sucCounts to the ($j$ – 1)th
          list
      end
  end

  if preCounts and sucCounts are not empty then
      STEP 2
      w = the first vertex in the last list of sucCounts               // heuristic 1
      path is empty
      while path does not contain w do
          w is added to path
          w = the first successor of w                                 // heuristic 2
      end
      loop = the part of path from the first occurrence of w to the end
```

```
        STEP 3
        selectedEdges = at least one edge which user has chosen to be removed
        for each edge in selectedEdges do
            x = the first vertex of the current edge
            y = the second vertex of the current edge
            x is removed from predecessors of y
            y is removed from successors of x
            x moves from the $i^{th}$ list (current position) of sucCounts to the $(i – 1)^{th}$ list
            y moves from the $j^{th}$ list (current position) of preCounts to the $(j – 1)^{th}$ list
        end
    end
end.
```

Pseudo code of algorithm RmL.

In the implementation of algorithm RmL, there are used two arrays, *preCounts* and *sucCounts*. *preCounts* is an array where on the i^{th} position, there is a list of vertices which have currently *i* predecessors (the i^{th} list), and *sucCounts* is an array where on the j^{th} position, there is a list of vertices which have currently *j* successors (the j^{th} list). At the beginning of algorithm RmL, these arrays are initialized by counting the number of predecessors and successors for each vertex. If a vertex is removed, the arrays are updated.

In step 2, two heuristics are used. The first heuristic is used for selecting a vertex from which we start to looking for a loop. It selects the first vertex in the last list of *sucCounts*, because the last list cannot be empty unlike others which can be (Empty lists of preCounts and sucCounts which are currently empty are removed between step 1 and step 2.). Heuristic 2 is used for selecting a successor, we go to. It selects the first successor of a vertex because we know that the vertex has at least one successor.

Properties of algorithm RmL

We have to prove that algorithm RmL removes all loops (cycles) in graph $G_{schedule}$ (proof of correctness) and that the algorithm is finite, i.e. removes those cycles in finite number of iterations (proof of finiteness) and each iteration is finite.

Proof of correctness of algorithm RmL: First of all, we prove three statements:
Statement 1: Step 1 removes all vertices which are not parts of cycles.
Statement 2: Step 2 finds a cycle.
Statement 3: Step 3 breaks the found cycle by removing of at least one edge of it.

Proof of statement 1: If we remove all vertices which have either no predecessor or no successor, only vertices which are parts of cycles are left. We must prove that two separated while loops in step 1 remove all vertices which are in the 0th lists of *preCounts* and *sucCounts* or which get to them during step 1. Now, we make a proof of correctness of the first loop. The proof of the second one is made in the same way.

Vertex *u* which is used in the pseudo code above is removed from *preCounts* and from *sucCounts*, because it is in the 0th list of *preCounts* (has no predecessor). However, the vertex might have some successors. Therefore, we must update locations of these tasks in *preCounts* (*u* was a predecessor of these tasks; so that, they lost a predecessor). That is the way, tasks which have some predecessors at the beginning can reach the 0th list of *preCounts* and they are removed too.

We have left to prove, that we can separate both while loops. We can see, in the first while loop, we do not change locations of tasks in *sucCounts*. Therefore, no vertex can reach the 0th list of *sucCounts* during the first while loop unless the vertex has been in the list before. Statement 1 has been proved.

Proof of statement 2: Statement 1 says that step 1 removes all vertices which are not parts of cycles. Thus, if any vertex left, it is a part of a cycle. There may be n vertices left, $n \in N \cup \{0\}$. Assume that we have already gone through $n–1$ vertices and up to now, we have not go to any of them twice. Vertex, where we are currently, must have at least one successor. This successor must be a vertex, we went through before, because we have only n vertices. Thus, we found a cycle. Statement 2 has been proved.

Proof of statement 3: User must select at least one edge from those which formed the found cycle. The selected edges are removed and the cycle exists no more. Statement 3 has been proved.

With using statement 3, it has been proved that algorithm RmL removes all cycles of graph $G_{schedule}$.

Proof of finiteness of algorithm RmL: We first prove the following statement:
Statement 4: Each iteration of algorithm RmL is finite.

Proof of statement 4: We have n vertices, $n \in N \cup \{0\}$. Therefore, while loops and for loops which are in step 1 are finite. Finiteness of step 2 has already been proved in statement 2 (above). In graph $G_{schedule}$, there are m edges, $m \in N \cup \{0\}$. Therefore, a number of iterations of for loop which is in step 3 is finite. Statement 4 has been proved.

In each iteration in step 3, at least one edge is removed there. Thus, a number of iterations of algorithm RmL is less or equal to *m* (the number of edges). With using it and statement 4, it has been proved that algorithm RmL removes all cycles in the finite number of iterations.

Correctness and finiteness of algorithm RmL have been proved.

3.2 Tasks in a temporal order (TmpO)

Two algorithms, TmpO-1 and TmpO-2, are described in this part. The reason is that it is complicated to prove that TmpO-2, which returns more compact schedules than TmpO-1 (see section 4.3), is finite. Thus, there are only proved properties of algorithm TmpO-1 below.

Before we describe algorithms TmpO-1 and TmpO-2, we define auxiliary variables for every tasks A, B such that A→B:

$diff_{A,B}$ ... equals $s_A + d_A – s_B$. If dependence $A \rightarrow B$ breaks constraint 1, $diff_{A,B} > 0$. Otherwise, $diff_{A,B} <= 0$.

$freeOnTheLeft_A$ ... equals $s_A – (s_{lp(A)} + d_{lp(A)})$. If task *A* has no predecessor, $freeOnTheLeft_A = s_A$.

For organizing tasks in a temporal order, we use iterative algorithms TmpO-1 and TmpO-2. The algorithms runs while the schedule does not satisfy constraint 1, i.e. while any dependence (=bad dependence) breaks constraint 1 (A, B, $A \rightarrow B$: $diff_{A,B} > 0$). In each iteration, the algorithms take one of these dependencies and modify it in such a way, that the dependence does not break constraint 1.

The important thing which is considered in the algorithms is that we do not want to spread out the schedule to the right unless it is necessary. Therefore, in the algorithms, there are two possibilities how to resolve inconsistency. It depends on the fact whether $freeOnTheLeft_A$ >= $diff_{A,B}$ (1) or not (2).

Algorithm TmpO-1:

(1) $freeOnTheLeft_A >= diff_{A,B}$. $s_A := s_A – diff_{A,B}$. The current value of $diff_{A,B} = 0$.

(2) freeOnTheLeftA < diffA,B. Then sA := sA – freeOnTheLeftA and sB := sA + dA. The current value of diffA,B = 0.

```
while a bad dependence exists do
    $AB$ = a bad dependence $A \rightarrow B$ with the least $s_B$
    $A$ = the first task of $AB$
    $B$ = the second task of $AB$

    $diff_{A,B} = s_A + d_A - s_B$
    if $A$ has at least one predecessor then
        $freeOnTheLeft_A = s_A - (s_{lp(A)} + d_{lp(A)})$
    else
        $freeOnTheLeft_A = s_A$
    end

    if $freeOnTheLeft_A$ >= $diff_{A,B}$ then
        (1)
        $s_A = s_A - diff_{A,B}$
    else
        (2)
        $s_A = s_A - freeOnTheLeft_A$
        $s_B = s_A + d_A$
    end
end.
```

Pseudo code of algorithm TmpO-1.

Algorithm TmpO-2:

(1) *freeOnTheLeft*$_A$ >= *diff*$_{A,B}$. $s_A := s_A - diff_{A,B}$. The current value of *diff*$_{A,B}$ = 0.

(2) *freeOnTheLeft*$_A$ < *diff*$_{A,B}$. $s_A := s_A - freeOnTheLeft_A$. Then, we calculate the current value of *diff*$_{A,B}$. *diff*$_{A,B}$ is still positive (> 0). Now, we have two possibilities how we can continue. It depends on a fact whether $s_A - \lfloor diff_{A,B} / 2 \rfloor < 0$ (a) or not (b). By both of them, we can break other dependencies. They are corrected in the following iterations.

   (a) $s_A - \lfloor diff_{A,B} / 2 \rfloor < 0$. Then $s_A := 0$ and $s_B := s_A + d_A$. The current value of *diff*$_{A,B}$ = 0.

   (b) $s_A - \lfloor diff_{A,B} / 2 \rfloor >= 0$. Then $s_A := s_A - \lfloor diff_{A,B} / 2 \rfloor$ and $s_B := s_B + \lceil diff_{A,B} / 2 \rceil$. The current value of *diff*$_{A,B}$ = 0. Thus, if *diff*$_{A,B}$ is odd integer, *B* is shifted more than *A* (the difference is 1).

```
while a bad dependence exists do
    $AB$ = a bad dependence $A \rightarrow B$ with the least $s_B$
    $A$ = the first task of $AB$
    $B$ = the second task of $AB$

    $diff_{A,B} = s_A + d_A - s_B$
    if $A$ has at least one predecessor then
        $freeOnTheLeft_A = s_A - (s_{lp(A)} + d_{lp(A)})$
    else
        $freeOnTheLeft_A = s_A$
    end

    if $freeOnTheLeft_A$ >= $diff_{A,B}$ then
        (1)
        $s_A = s_A - diff_{A,B}$
    else
        (2)
        $s_A = s_A - freeOnTheLeft_A$
        $diff_{A,B} = s_A + d_A - s_B$

        if $s_A$ =< $\lfloor diff_{A,B} / 2 \rfloor$ then
            (a)
            $s_A = 0$
            $s_B = s_A + d_A$
        else
            (b)
            $s_A = s_A - \lfloor diff_{A,B} / 2 \rfloor$
            $s_B = s_B + \lceil diff_{A,B} / 2 \rceil$
        end
    end
end.
```

Pseudo code of algorithm TmpO-2.

We can see that both algorithms, TmpO-1 and TmpO-2, are different only in policy (2). Algorithm TmpO-2 can shift task A more to the left than algorithm TmpO-1. Therefore, unlike algorithm TmpO-1, algorithm TmpO-2 can break dependencies $C \rightarrow A$.

Properties of algorithm TmpO-1

We have to prove that algorithm TmpO-1 organizes tasks in a temporal order (proof of correctness) and that the algorithm is finite, i.e. organizes tasks in a temporal order in finite number of iterations (proof of finiteness) and each iteration is finite.

Proof of correctness of algorithm TmpO-1: First of all, we prove the following statement:
Statement 1: Each iteration of algorithm TmpO-1 corrects a bad dependence.

Proof of statement 1: All possibilities, how a dependence can be corrected during an iteration, ends with $diff_{A,B} = 0$. It means that the dependence satisfies constraint 1 at the end of the iteration. Statement 1 has been proved.

With using statement 1 and the fact that the algorithm does no finish until a bad dependence exists, it has been proved that algorithm TmpO-1 organizes tasks in a temporal order.

Proof of finiteness of algorithm TmpO-1: We first prove two statements:
Statement 2: Each iteration of algorithm TmpO-1 is finite.
Statement 3: No dependence $C \rightarrow A$ can be broken.

Proof of statement 2: In algorithm TmpO-1 there is no loop inside the while loop. Thus, each iteration must be finite. Statement 2 has been proved.

Proof of statement 3: In both policies (1) and (2) of algorithm TmpO-1, task *A* is shifted to the left not more than about a value of $freeOnTheLeft_A$. With using the definition of the auxiliary variable $freeOnTheLeft_A$ (see above), statement 3 has been proved.

With using statement 1 and statement 3 we have proved that in each iteration a bad dependence $A \rightarrow B$ is corrected and no dependence $C \rightarrow A$ is broken. It means that dependence $A \rightarrow B$ is corrected only once because we correct dependencies from the left to the right (bad dependency with the least s_B is corrected first in each iteration). From these facts, from the fact that a schedule has $m \in N \cup \{0\}$ dependencies and with using statement 2, it has been proved that algorithm TmpO-1 is finite.

Correctness and finiteness of algorithm TmpO-1 have been proved.

3.3 Satisfying constraints 2 (and 1) (C2C1)

Before we describe algorithm C2C1, we define an auxiliary variable for every tasks A, B such that $A \rightarrow B$ and for every tasks A, B and resource R such that $A(R)$, $B(R)$:

> $diff_{A,B}$ ... equals $s_A + d_A - s_B$. If dependence $A \rightarrow B$ breaks constraint 1 or 2, $diff_{A,B} > 0$. Otherwise, $diff_{A,B} <= 0$.

Iterative algorithm which is used in this phase is similar to algorithms TmpO-1 and TmpO-2. Algorithm C2C1 runs until a schedule satisfies constraints 1 and 2 at the same time (until it is feasible), i.e. until a pair of tasks A, B such that $diff_{A,B} > 0$ exists. In each iteration there is selected one of them (one of pairs of tasks) and it is modified in such a way, that it satisfies the constraint which it has broken before. Pairs which break constraint 1 are modified at first.

Both kinds of pairs, those which break constraint 1 as well as those which break constraint 2, are modified the same way: $s_B = s_B + diff_{A,B}$. Thus, $A \rightarrow B$ is now correct. In comparison with algorithm TmpO, no dependence which task *A* is part of can be broken by these modifications, because *A* stays on the same place. By shifting *B* to the right, dependencies in which task *B* is present can be broken. These dependencies are corrected in other iterations of algorithm C2C1.

```
while a bad dependence exists or a conflict on a resource exists do
   if a bad dependence exists then
         AB = a bad dependence $A \rightarrow B$ with the least $s_B$        // heuristic 1
   else
         AB = a conflict of A, B on resource R, A(R), B(R) with the least $s_A$ and
         the least $s_B$                                            // heuristic 2
   end

   A = the first task of AB
   B = the second task of AB
   $diff_{A,B} = s_A + d_A - s_B$

   $s_B = s_B + diff_{A,B}$
end.
```

Pseudo code of algorithm C2C1.

In the pseudo code of algorithm C2C1 (above), heuristic 1 is used for selecting among bad dependencies. It selects such a dependence $A \rightarrow B$, where s_B has the smallest value. If we do not use this heuristic, the algorithm C2C1 will probably have more iterations, because a dependence can be solved more times. In the pseudo code, heuristic 2 is used as well. It selects among inconsistencies on resources. The heuristic takes such a conflict between *A*, *B* where s_A has the smallest value. If there are more pairs with the same value of s_A, then it selects that one which has the smallest value of s_B. Heuristic 2 is used because we want to have as few as possible iterations. If we select s_A with the smallest value, conflicts which break the constraint are modified only once. If we select s_B with the smallest value, it is more likely that dependences in which *B* is part of will not be broken. If we do not use the second condition, we probably broken more dependencies.

Properties of algorithm C2C1

We have to prove that algorithm TmpO-1 organizes tasks in such a way that both constraints (1 and 2) are satisfied at the same time (proof of correctness) and that the algorithm is finite, i.e. organizes tasks in finite number of iterations (proof of finiteness) and each iteration is finite.

Proof of correctness of algorithm C2C1: First of all, we prove the following statement:
Statement 1: Each iteration of algorithm C2C1 corrects an inconsistence.

Proof of statement 1: First, an inconsistence (a bad dependence or a conflict on a resource) is selected. Then, task B is shifted just behind task A. So that at the end of iteration, the inconsistence is corrected. Statement 1 has been proved.

With using statement 1 and the fact that the algorithm do no finish until any inconsistence exists, it has been proved that algorithm C2C1 organizes tasks in such a way that both constraints (1 and 2) are satisfied at the same time.

Proof of finiteness of algorithm C2C1: We first prove three statements:
Statement 2: Each iteration of algorithm C2C1 is finite.
Statement 3: In each iteration, s_B is increased.
Statement 4: Tasks are shifted only to the right.

Proof of statement 2: In algorithm C2C1 there is no loop inside the while loop. Thus, each iteration must be finite. Statement 2 has been proved.

Proof of statement 3: In each iteration, there is selected one inconsistence (a bad dependence or a conflict of a resource) between two tasks A, B such that $diff_{A,B} > 0$. s_B is modified only in the end of an iteration in this way: $s_B = s_B + diff_{A,B}$. Thus, statement 3 has been proved.

Proof of statement 4: In the algorithm, we do not shift any task to the left and with using statement 3 we know that tasks are shifted to the right. Statement 4 has been proved.

We know that a schedule has n tasks, $n \in N \cup \{0\}$, and that $d_T \in N \cup \{0\}$ for each task in the schedule. We consider that a schedule, that we reschedule, is feasible only when all tasks are performed in a different time, i.e. it must be performed not more than one task at the time. So that, with using statement 3 and statement 4, we can shift tasks to such places in finite number of iterations. Since the number of iterations is the finite and with using statement 2, we prove that algorithm C2C1 is finite.

Correctness and finiteness of algorithm C2C1 have been proved.

3.4 Properties of rescheduling algorithm

Rescheduling algorithm is formed by tree phases: removing of all loops (RmL), tasks in a temporal order (TmpO) and satisfying constraints 2 and 1 (C2C1). Proofs of their correctness and finiteness were done with their description.

Because algorithms of all phases are finite and correct, rescheduling algorithm must be finite and correct too.

Chapter 4

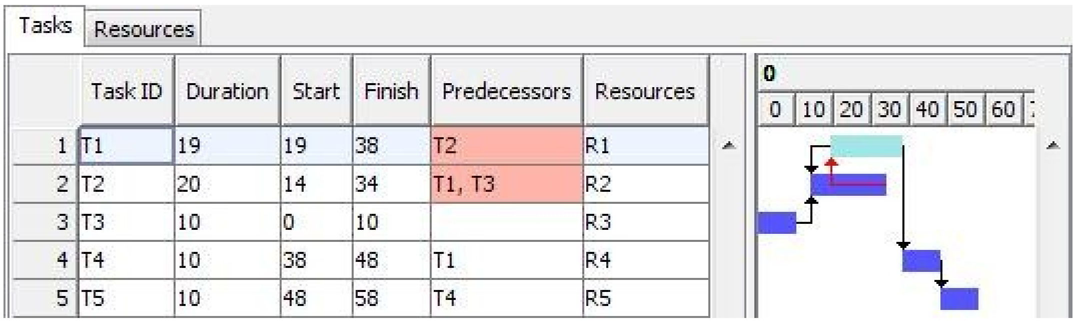
Demonstration of the rescheduling algorithm

In this chapter, we demonstrate the applicability of the rescheduling algorithm on some examples. For this reason, program GanttViewer was developed. Rescheduling process in the program can be stopped at any time and user gets a schedule with all changes which have already been made. This can be used if the algorithm seems to run too long. User stops the algorithm, makes required changes in the current schedule and starts the algorithm again.

We demonstrate the algorithm on three schedules: Schedule8.xml, Schedule5.txt and Schedule7.xml (all of them are at the attached CD in directory test_input). Using schedule Schedule8.xml, we show the first two phases of the algorithm, removing of all loops and tasks in a temporal order (both algorithms, TmpO-1 and TmpO-2, are demonstrated), and using schedule Schedule5.txt, we demonstrate the third phase, satisfying constraint 2 (and 1). Finally, we test the algorithm on schedule Schedule7.xml, which is larger than the previous ones. Results of the test show us, whether the algorithm is efficient on larger data.

4.1 Demonstration of phases 1 and 2

On schedule Schedule8.xml, we demonstrate the first two phases, removing of all loops and tasks in a temporal order. We also compare algorithms TmpO-1 and TmpO-2.

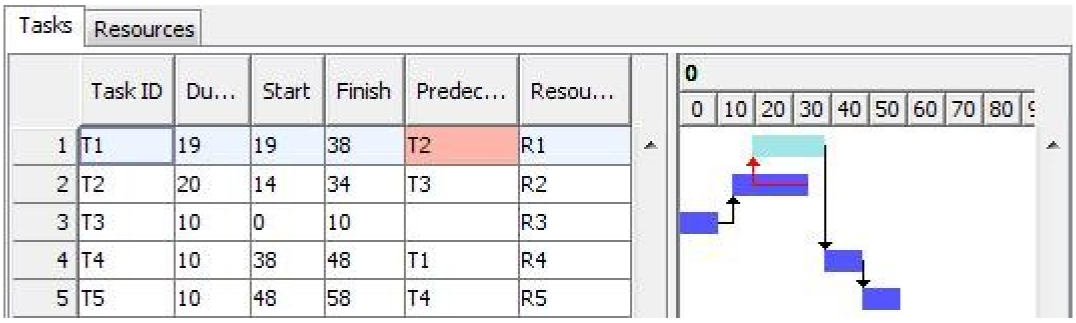


	Task ID	Duration	Start	Finish	Predecessors	Resources
1	T1	19	19	38	T2	R1
2	T2	20	14	34	T1, T3	R2
3	T3	10	0	10		R3
4	T4	10	38	48	T1	R4
5	T5	10	48	58	T4	R5

figure 4.1 schedule Schedule8.xml before rescheduling

In figure 4.1, we see that each task has it's own resource on which it is performed. Therefore, the schedule satisfies constraint 2 all the time.

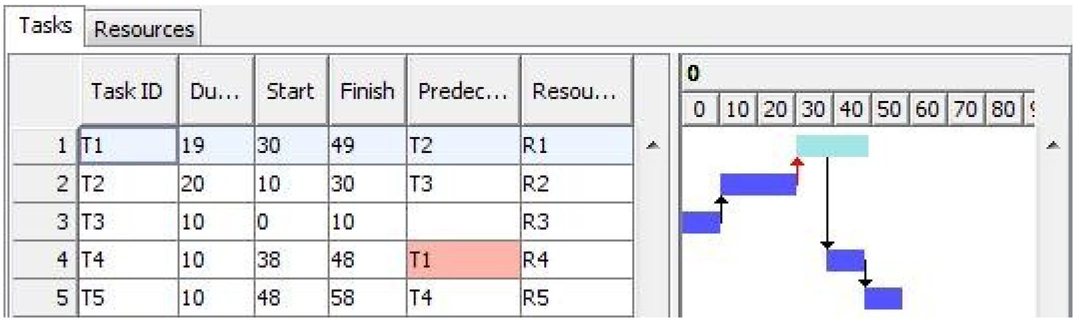
After a start of the rescheduling algorithm, a loop formed by dependencies *T1→T2*, *T2→T1* (see figure 4.1). We remove the dependence *T1→T2*.



	Task ID	Du...	Start	Finish	Predec...	Resou...
1	T1	19	19	38	T2	R1
2	T2	20	14	34	T3	R2
3	T3	10	0	10		R3
4	T4	10	38	48	T1	R4
5	T5	10	48	58	T4	R5

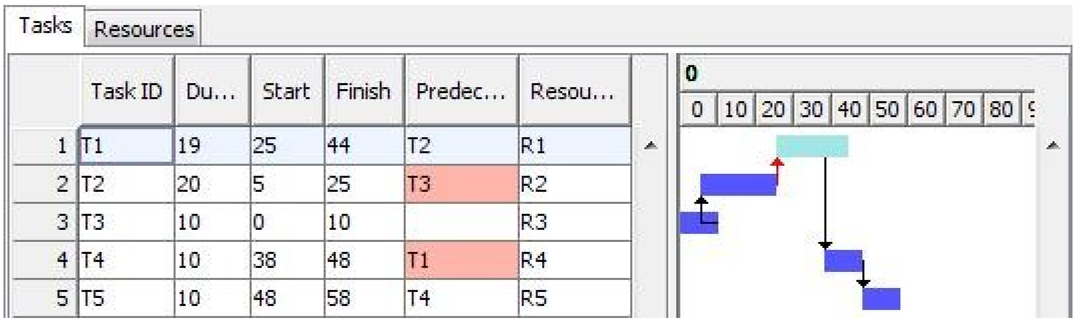
figure 4.2 schedule Schedule8.xml after removing of the dependence T1→T2

In figure 4.2, we see that no loop is in the schedule now. Thus, the phase of removing of loops has finished and the phase of a temporary order of tasks starts. In the schedule, there is only one dependence, which breaks constraint 1. It is the dependence *T2→T1*. In both algorithms TmpO-1 and TmpO-2, two auxiliary variables are calculated, $diff_{T2,T1}$ and $freeOnTheLeft_{T2}$ (see chapter 3.2). Variable $diff_{T2,T1} = s_{T2} + d_{T2} – s_{T1}$ = 14 + 20 – 19 = 15 and $freeOnTheLeft_{T2} = s_{T2} – (s_{T3} + d_{T3})$ = 14 – (0 + 10) = 4. In both algorithms, we use policy (2) (see chapter 3.2).



	Task ID	Du...	Start	Finish	Predec...	Resou...
1	T1	19	30	49	T2	R1
2	T2	20	10	30	T3	R2
3	T3	10	0	10		R3
4	T4	10	38	48	T1	R4
5	T5	10	48	58	T4	R5

figure 4.3a schedule Schedule8.xml after the first application of policy (2) of algorithm TmpO-1



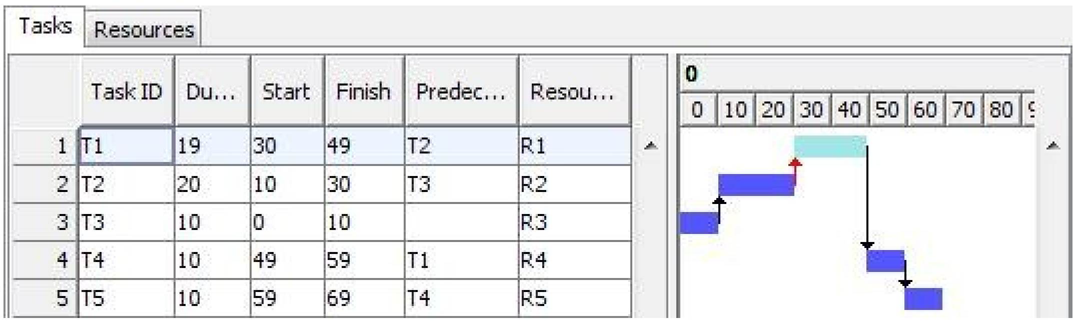
	Task ID	Du...	Start	Finish	Predec...	Resou...
1	T1	19	25	44	T2	R1
2	T2	20	5	25	T3	R2
3	T3	10	0	10		R3
4	T4	10	38	48	T1	R4
5	T5	10	48	58	T4	R5

figure 4.3b schedule Schedule8.xml after the first application of policy (2) of algorithm TmpO-2

We see that algorithm TmpO-2 shifts task *T2* more to the left (see figure 4.3b) than algorithm TmpO-1 (see figure 4.3a). Therefore, in algorithm TmpO-1, there is broken only the dependence *T1→T4* on the right from the dependence *T2→T1*. In algorithm TmpO-2, there is also broken the dependence *T3→T2* on the left.

If we simulate algorithm TmpO-1 further, we see that only dependencies on the right from *T2→T1* are broken just one time. So that the algorithm finishes after another two iterations.

If we simulate algorithm TmpO-2 further, we see that each dependence is broken at least at one time. However, after some iterations of the algorithm (more than two), tasks *T1*, *T2*, *T4* and *T5* are shifted enough to the right and the schedule satisfies constraint 1.



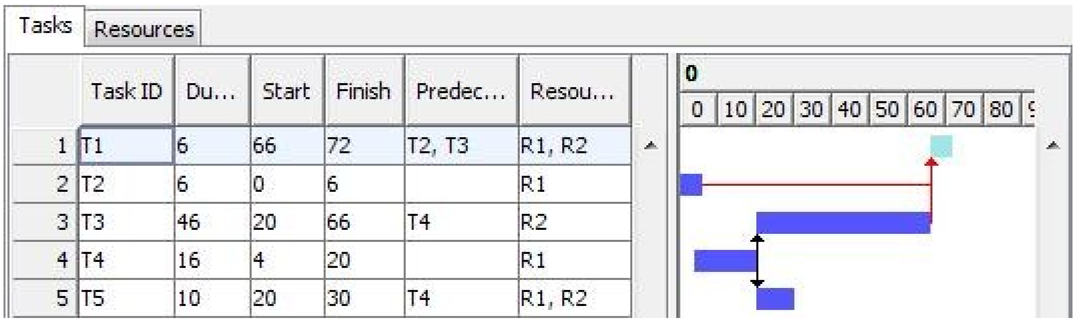
	Task ID	Du...	Start	Finish	Predec...	Resou...
1	T1	19	30	49	T2	R1
2	T2	20	10	30	T3	R2
3	T3	10	0	10		R3
4	T4	10	49	59	T1	R4
5	T5	10	59	69	T4	R5

figure 4.4 schedule Schedule8.xml satisfies constraint 1

Since the schedule must satisfy constraint 2 (each task is performed by different resource), in figure 4.4, we see the feasible schedule. Therefore, both rescheduling algorithms have finished. Their results are the same. We see that the schedule is as compact as it is possible. So that, algorithms solved conflicts locally and they return a compact schedule. That is what we want.

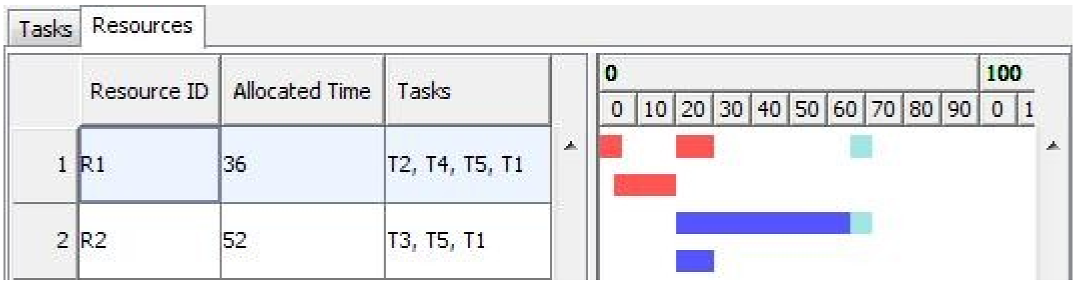
4.2 Demonstration of phase 3

On schedule Schedule5.txt, we demonstrate the third phase, satisfying constraints 2 (and 1).



	Task ID	Du...	Start	Finish	Predec...	Resou...
1	T1	6	66	72	T2, T3	R1, R2
2	T2	6	0	6		R1
3	T3	46	20	66	T4	R2
4	T4	16	4	20		R1
5	T5	10	20	30	T4	R1, R2

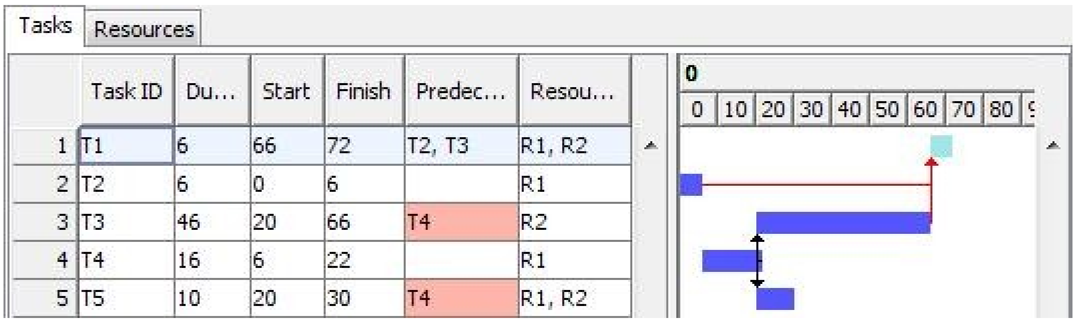
figure 4.5 schedule Schedule5.txt before rescheduling – tasks



	Resource ID	Allocated Time	Tasks
1	R1	36	T2, T4, T5, T1
2	R2	52	T3, T5, T1

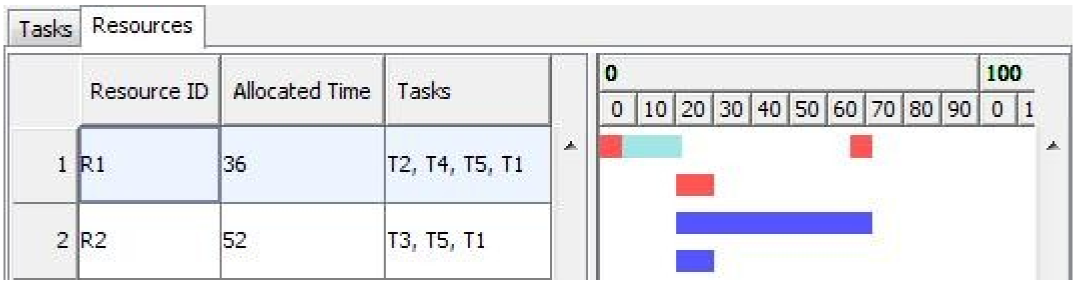
figure 4.6 schedule Schedule5.txt before rescheduling – resources

In schedule Schedule5.txt, there is no loop (see figure 4.5). So that, the schedule satisfies constraint 1. In figure 4.6, we see that on both resources, *R1* and *R2*, there are conflicts of tasks. Thus, the schedule does not satisfy constraint 2. According to algorithm C2C1, the first pair of tasks which breaks constraint 2 is *T2*, *T4* on resource *R1*. s_{T2} is less than s_{T4}, so that, the algorithm shifts *T4* behind *T2* (see figure 4.8).



	Task ID	Du...	Start	Finish	Predec...	Resou...
1	T1	6	66	72	T2, T3	R1, R2
2	T2	6	0	6		R1
3	T3	46	20	66	T4	R2
4	T4	16	6	22		R1
5	T5	10	20	30	T4	R1, R2

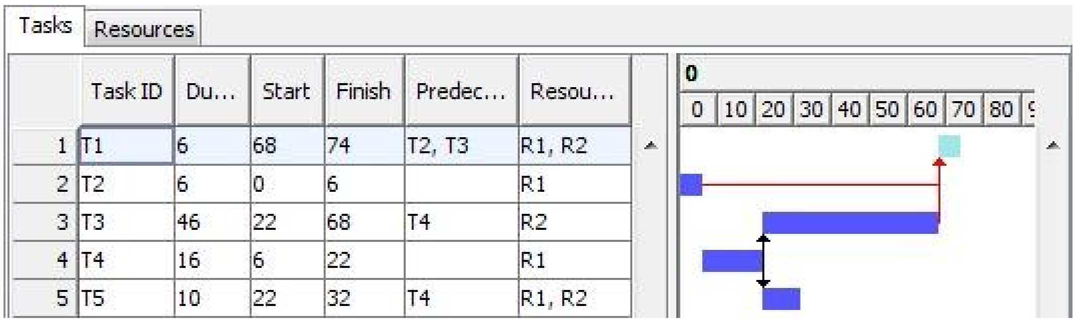
figure 4.7 schedule Schedule5.txt after shifting task T4 behind task T2 – tasks



	Resource ID	Allocated Time	Tasks
1	R1	36	T2, T4, T5, T1
2	R2	52	T3, T5, T1

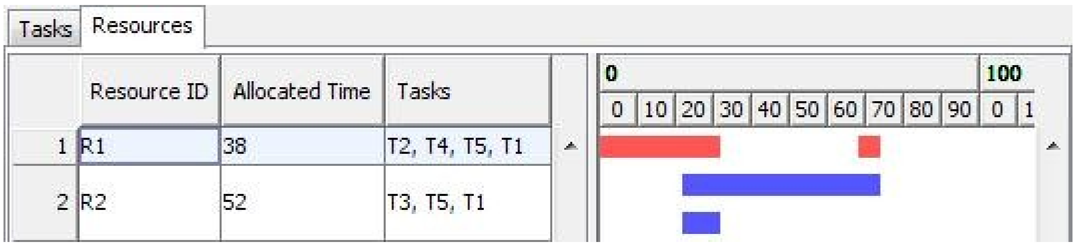
figure 4.8 schedule Schedule5.txt after shifting task T4 behind task T2 – resources

By shifting task *T4* to the right, there are two dependencies in the schedule which break constraint 1: *T4→T3* and *T4→T5* (see figure 4.7). In the following three iterations, the algorithm shifts tasks *T3*, *T1* (*T3→T1* will break constraint 1) and *T5* in turn to the right.



	Task ID	Du...	Start	Finish	Predec...	Resou...
1	T1	6	68	74	T2, T3	R1, R2
2	T2	6	0	6		R1
3	T3	46	22	68	T4	R2
4	T4	16	6	22		R1
5	T5	10	22	32	T4	R1, R2

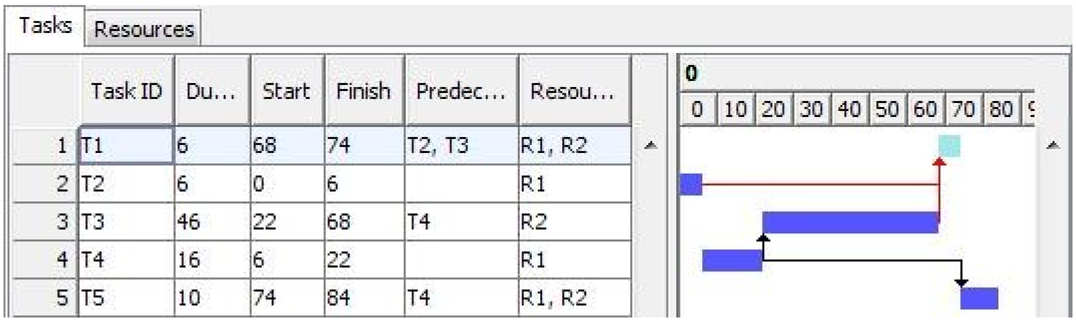
figure 4.9 schedule Schedule5.txt after shifting tasks T3, T1, T5 to the right – tasks



	Resource ID	Allocated Time	Tasks
1	R1	38	T2, T4, T5, T1
2	R2	52	T3, T5, T1

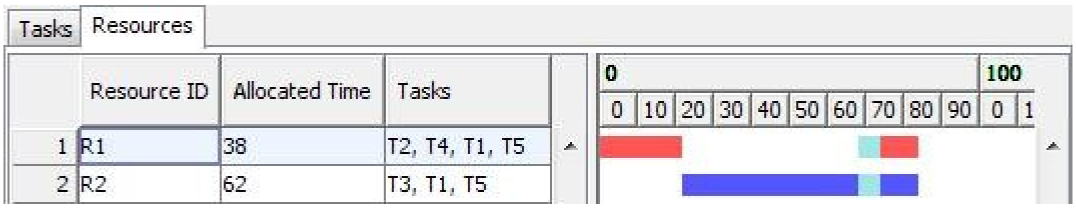
figure 4.10 schedule Schedule5.txt after shifting tasks T3, T1, T5 to the right – resources

The current schedule satisfies constraint 1 again. On resource *R2*, there are still tasks which break constraint 2. Thus, the algorithm repeats the previous steps for a pair of tasks *T3*, *T5*. Since *T5* will be shifted behind *T3* (*T5* has no successor), the steps are also repeated for pair *T1*, *T5*.



	Task ID	Du...	Start	Finish	Predec...	Resou...
1	T1	6	68	74	T2, T3	R1, R2
2	T2	6	0	6		R1
3	T3	46	22	68	T4	R2
4	T4	16	6	22		R1
5	T5	10	74	84	T4	R1, R2

figure 4.11 the final schedule Schedule5.txt – tasks



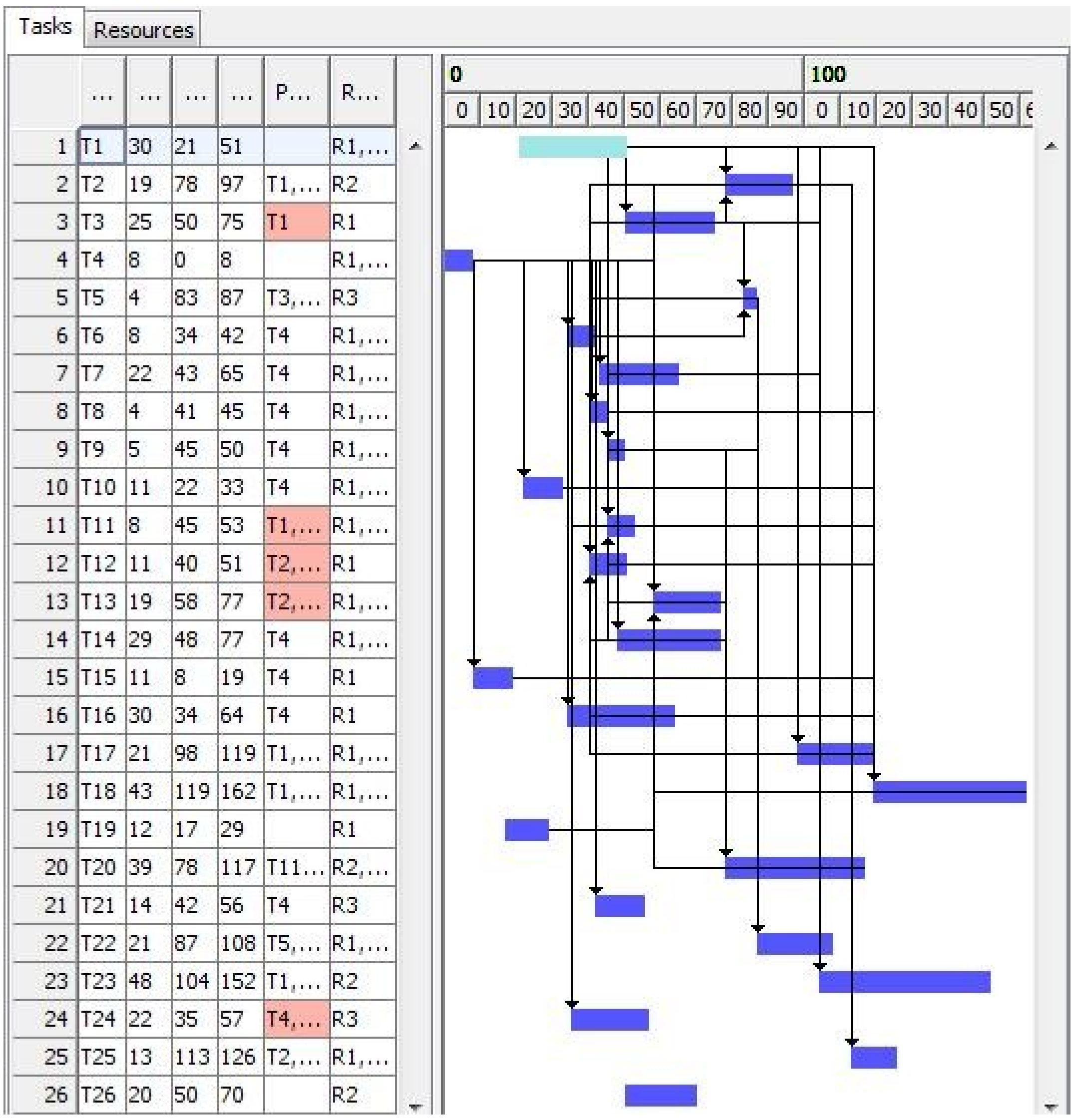
	Resource ID	Allocated Time	Tasks
1	R1	38	T2, T4, T1, T5
2	R2	62	T3, T1, T5

figure 4.12 the final schedule Schedule5.txt – resources

We can see the final schedule in figures 4.11 and 4.12. We see that in figure 4.12, tasks tie together closely (free space on resource *R1* is caused by dependencies). The schedule is compact. If we compare figure 4.5 and 4.10, we see that tasks except *T5* are almost at the same place.

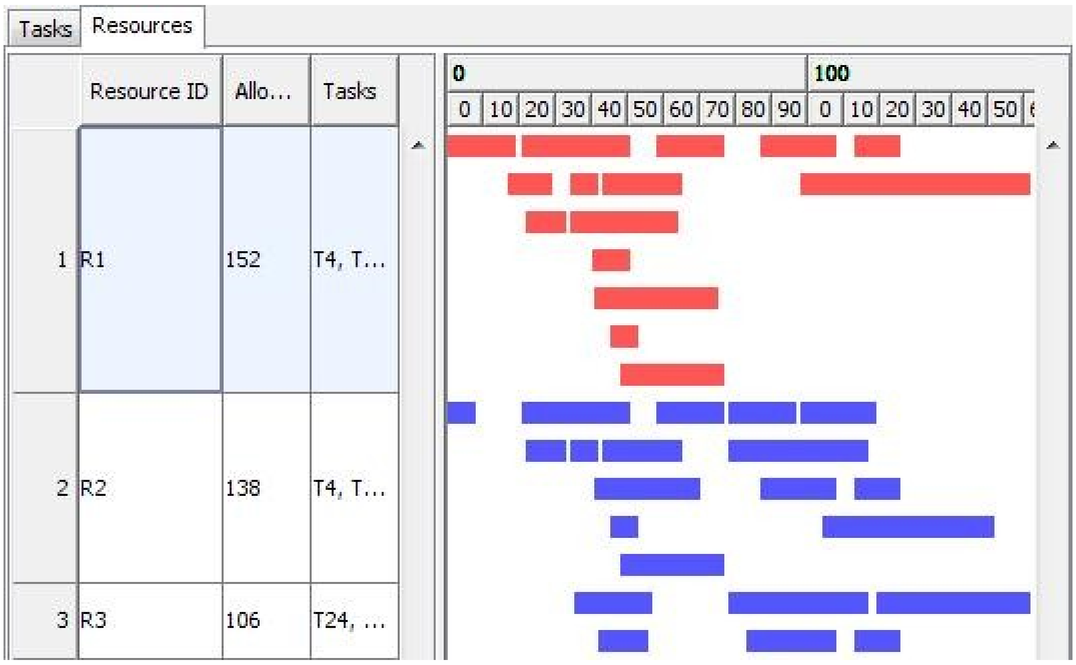
4.3 Test on large data

On schedule Schedule7.xml, we test whether the designed algorithm is efficient on larger data. The schedule has 26 tasks, 3 resources and 78 dependencies.


Tasks | Resources

	...	...	...	...	P...	R...
1	T1	30	21	51		R1,...
2	T2	19	78	97	T1,...	R2
3	T3	25	50	75	T1	R1
4	T4	8	0	8		R1,...
5	T5	4	83	87	T3,...	R3
6	T6	8	34	42	T4	R1,...
7	T7	22	43	65	T4	R1,...
8	T8	4	41	45	T4	R1,...
9	T9	5	45	50	T4	R1,...
10	T10	11	22	33	T4	R1,...
11	T11	8	45	53	T1,...	R1,...
12	T12	11	40	51	T2,...	R1
13	T13	19	58	77	T2,...	R1,...
14	T14	29	48	77	T4	R1,...
15	T15	11	8	19	T4	R1
16	T16	30	34	64	T4	R1
17	T17	21	98	119	T1,...	R1,...
18	T18	43	119	162	T1,...	R1,...
19	T19	12	17	29		R1
20	T20	39	78	117	T11...	R2,...
21	T21	14	42	56	T4	R3
22	T22	21	87	108	T5,...	R1,...
23	T23	48	104	152	T1,...	R2
24	T24	22	35	57	T4,...	R3
25	T25	13	113	126	T2,...	R1,...
26	T26	20	50	70		R2

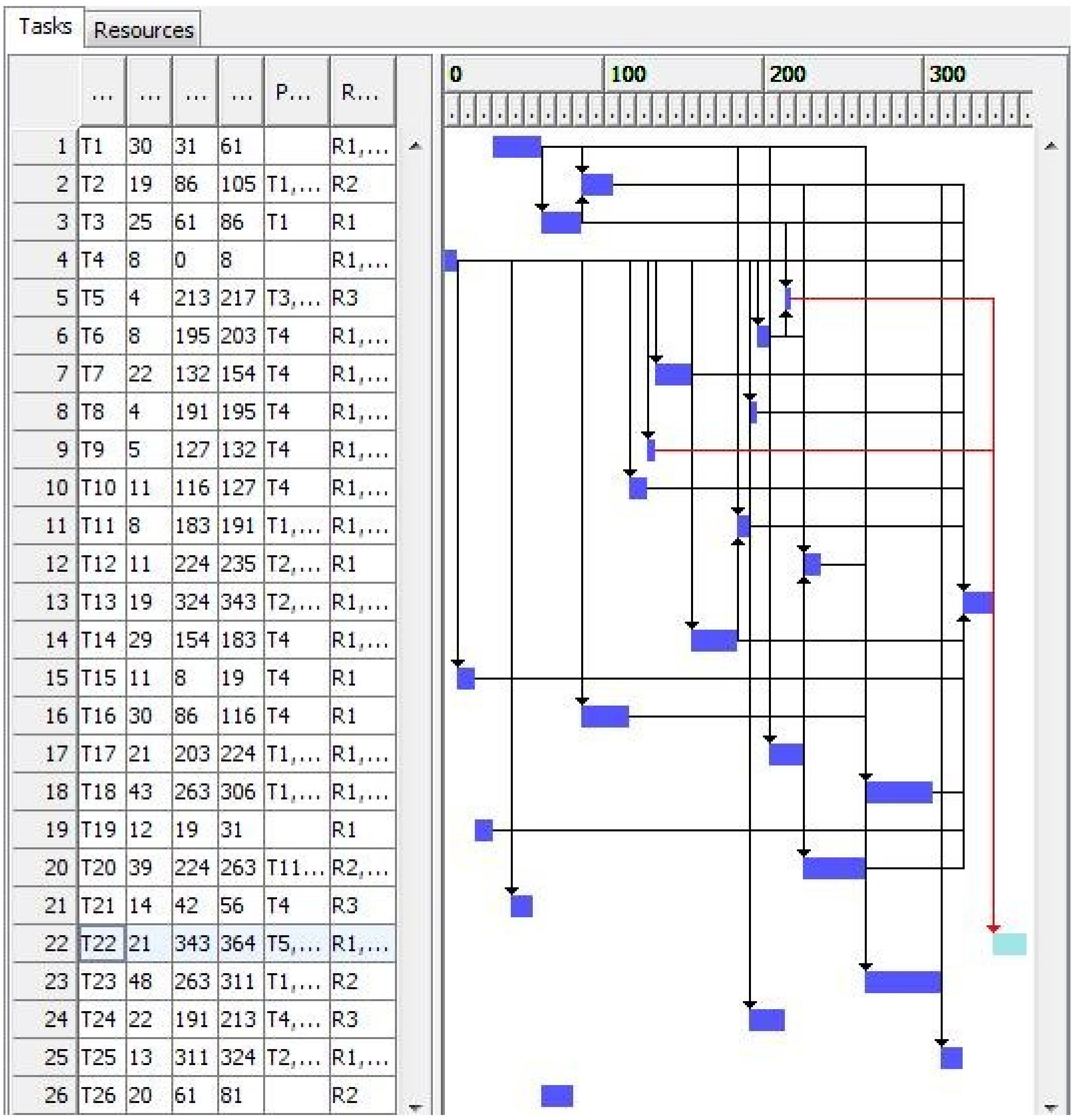
figure 4.13 schedule Schedule7.xml before rescheduling – tasks



	Resource ID	Allo...	Tasks
1	R1	152	T4, T...
2	R2	138	T4, T...
3	R3	106	T24, ...

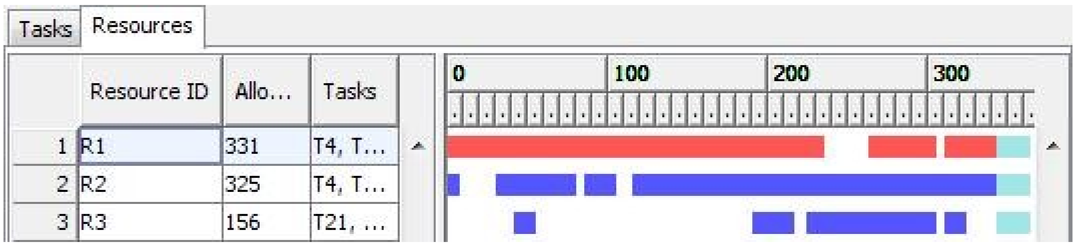
figure 4.14 schedule Schedule7.xml before rescheduling – resources

In the first phase (removing of all loops), we remove for example *T12→T11*, *T12→T17*, *T13→T11* and *T13→T20*. No other dependence must be removed, because there is no loop in the current schedule.



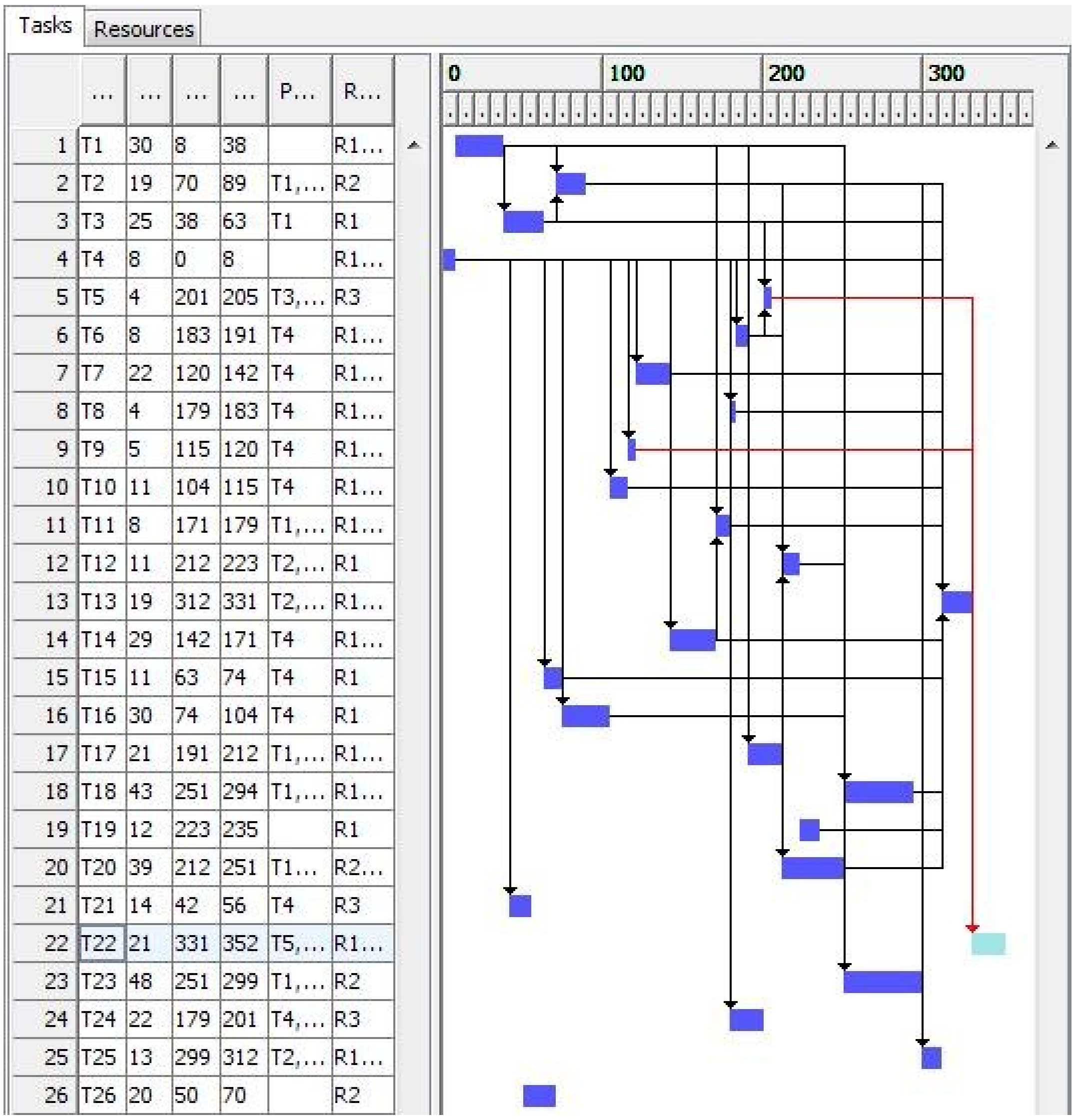
	…	…	…	…	P…	R…
1	T1	30	31	61		R1,…
2	T2	19	86	105	T1,…	R2
3	T3	25	61	86	T1	R1
4	T4	8	0	8		R1,…
5	T5	4	213	217	T3,…	R3
6	T6	8	195	203	T4	R1,…
7	T7	22	132	154	T4	R1,…
8	T8	4	191	195	T4	R1,…
9	T9	5	127	132	T4	R1,…
10	T10	11	116	127	T4	R1,…
11	T11	8	183	191	T1,…	R1,…
12	T12	11	224	235	T2,…	R1
13	T13	19	324	343	T2,…	R1,…
14	T14	29	154	183	T4	R1,…
15	T15	11	8	19	T4	R1
16	T16	30	86	116	T4	R1
17	T17	21	203	224	T1,…	R1,…
18	T18	43	263	306	T1,…	R1,…
19	T19	12	19	31		R1
20	T20	39	224	263	T11…	R2,…
21	T21	14	42	56	T4	R3
22	T22	21	343	364	T5,…	R1,…
23	T23	48	263	311	T1,…	R2
24	T24	22	191	213	T4,…	R3
25	T25	13	311	324	T2,…	R1,…
26	T26	20	61	81		R2

figure 4.15a the final schedule Schedule7.xml – tasks (TmpO-1 used)



	Resource ID	Allo…	Tasks
1	R1	331	T4, T…
2	R2	325	T4, T…
3	R3	156	T21, …

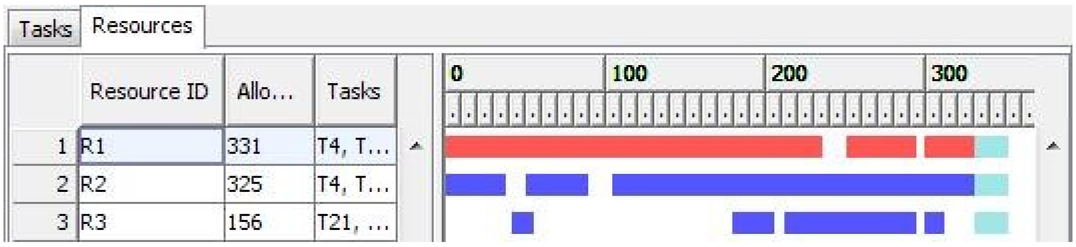
figure 4.16a the final schedule Schedule7.xml – resources (TmpO-1 used)



Tasks | Resources

	...	...	...	...	P...	R...
1	T1	30	8	38		R1...
2	T2	19	70	89	T1,...	R2
3	T3	25	38	63	T1	R1
4	T4	8	0	8		R1...
5	T5	4	201	205	T3,...	R3
6	T6	8	183	191	T4	R1...
7	T7	22	120	142	T4	R1...
8	T8	4	179	183	T4	R1...
9	T9	5	115	120	T4	R1...
10	T10	11	104	115	T4	R1...
11	T11	8	171	179	T1,...	R1...
12	T12	11	212	223	T2,...	R1
13	T13	19	312	331	T2,...	R1...
14	T14	29	142	171	T4	R1...
15	T15	11	63	74	T4	R1
16	T16	30	74	104	T4	R1
17	T17	21	191	212	T1,...	R1...
18	T18	43	251	294	T1,...	R1...
19	T19	12	223	235		R1
20	T20	39	212	251	T1...	R2...
21	T21	14	42	56	T4	R3
22	T22	21	331	352	T5,...	R1...
23	T23	48	251	299	T1,...	R2
24	T24	22	179	201	T4,...	R3
25	T25	13	299	312	T2,...	R1...
26	T26	20	50	70		R2

figure 4.15b the final schedule Schedule7.xml – tasks (TmpO-2 used)



Tasks | Resources

	Resource ID	Allo...	Tasks
1	R1	331	T4, T...
2	R2	325	T4, T...
3	R3	156	T21, ...

figure 4.16b the final schedule Schedule7.xml – resources (TmpO-2 used)

In figures 4.15a, 4.15b, 4.16a and 4.16b, we can see that the final schedules are feasible (in figure 4.15a (4.15b), there is no cell with red background color and in figure 4.16a (4.16b), there is one row per resource). However, the final schedule returned by algorithm using TmpO-1 is less compact than the second final schedule (the latest task T22 finishes later in the first schedule). If we look at figure 4.16a (4.16b), we see that resource R1 use more than 90% of it's time to perform tasks (the schedules are scheduled from 0 to 364 and from 0 to 352 time units and resource R1 perform tasks for 331 time units). Thus, both rescheduling algorithms return very compact schedules. If we compare figures 4.13 and 4.15a (4.15b), we see that the schedules has been dramatically spread out to the right. It is a consequence of a fact that there are 26 tasks and only 3 resources.

If we measure how long the algorithm runs in program GanttViewer we find out that in the first phase (removing of all loops), dependencies which form loops are shown in tenths of second (technical details about machine are below in section 4.4). Other phases take few seconds.

4.4 Summary

In this chapter, we demonstrated the designed algorithm in program GanttViewer. The program works fine with both small and large data (tens of tasks). Rescheduling takes few seconds (we cannot calculate time which user spend by selecting dependencies which should be removed).

The program was demonstrated and tested on a machine with Intel Core 2 1.83GHz, 1GB RAM, Windows Vista^{TM} Business with Service Pack 1, Java Development Kit 6 (Sun Microsystems).

Conclusion

This bachelor thesis deal with problems of interactive rescheduling in Gantt charts. In the work, we designed two rescheduling algorithms. Both of them have been realized. However, there have not been proved properties of the second algorithm which returns more compact schedule than the first one. We demonstrated both algorithms and compared results.

Program GanttViewer, a part of the thesis, implements both algorithms. The program has a graphic user interface so that, it can be used for creating and editing schedules in Gantt charts. Using the program, user can compare algorithms himself.

In the future, this thesis can be used as a draft for another work because the second rescheduling algorithm has not been proved. The development of rescheduling algorithm is also possible, e.g. the algorithm could be probably applied for rescheduling schedules with cumulative resources. These resources can perform more than one task at a time.

Literature

[1] Cesta A., Oddi A., Policella N., Smith S. F.: *Boosting the Performance of Iterative Flattening Search*, Springer Berlin / Heidelberg, August 2007

[2] Cesta A., Oddi A., Policella N., Smith S. F.: *Hybrid Variants for Iterative Flattening Search*, Springer Berlin / Heidelberg, May 2008

[3] Müller T.: *Interactive Timetabling*, Master Thesis, KTIML MFF UK, Prague, September 2001

Appendix A

User guide

A Preface

Program GanttViewer is designed for **displaying, modifying and optimizing** project schedules in Gantt charts.

In the program, the user can create new schedules. He can load (save) schedules from (to) text files with the specified format (TXT format, XML format) as well. The user can add or remove tasks or resources or modify them at all. Finally, the user can use rescheduling function to get the feasible schedule.

Program GanttViewer is independent on a computing platform. It means that it runs on any operation system.

Chapter A.1

Start and exit

A.1.1 Start

Before the user starts program GanttViewer, JAVA SDK or JRE is needed to be install. He can download it on http://developers.sun.com/downloads/.
He has to execute "**GanttViewer.jar**" file to start program GanttViewer.

After the start of the program, the main window appears. The user can display and modify a project schedule in it.

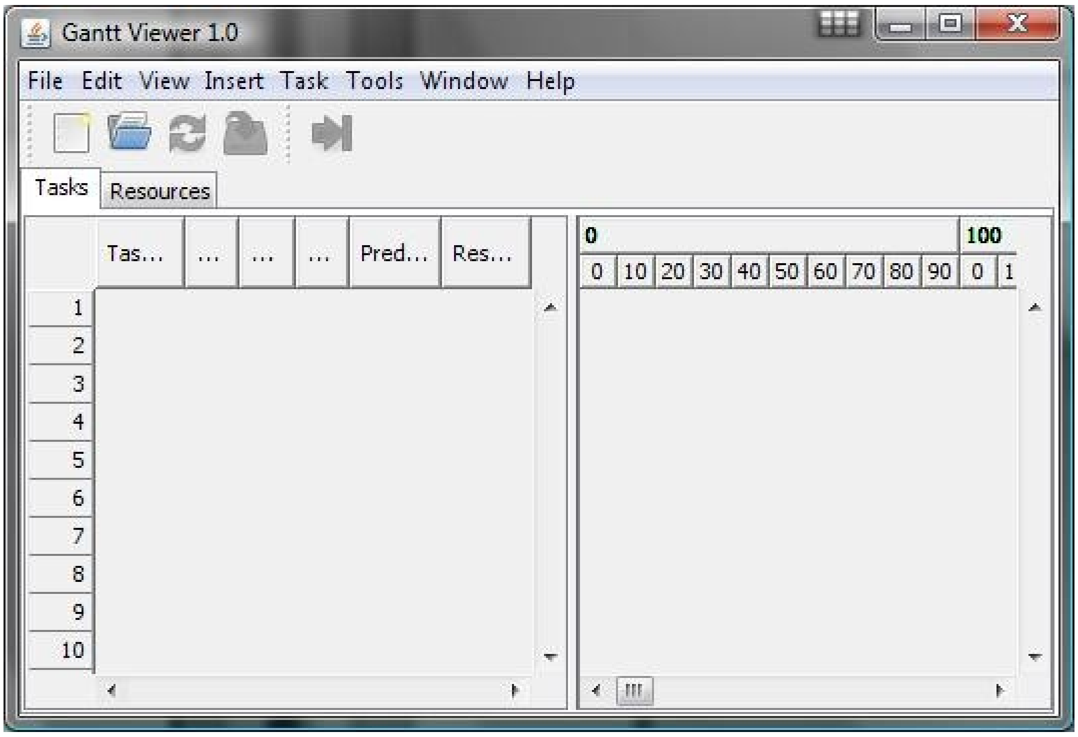


figure 1.1 the main window of program GanttViewer

A.1.2 Exit

If the user wants to exit from the program, click on *File > Exit* or a cross in the top right corner or type *Alt+F4*. Before the program terminates, all open schedules are checked whether they are saved. If a schedule is not saved, the program offers the user to save it.

Chapter A.2

Work area

A.2.1 Menu bar and toolbars

If the user wants to create a new schedule, to switch to “Resources” pane or to remove a selected task, he can do that using a menu bar or toolbars. There are details about them below.

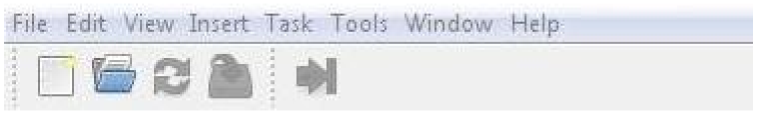


figure 2.1 | *File toolbar* | *Tools toolbar*

File

> *New Schedule...*	creates a new schedule. (see Creating schedules)
> *Open Schedule...*	loads a schedule from a file. (see Opening schedules)
> *Reload*	loads the shown schedule again. The user is asked whether he wants to save unsaved changes. (see Opening schedules)
> *Close...*	closes the shown schedule. The user is asked whether he wants to save unsaved changes. (see Closing schedules)
> *Schedule Properties*	for changing properties of the shown schedule. (see Modifying schedule properties)
> *Save*	saves the shown schedule.

	(see Saving schedules)
> *Save As...*	saves the shown schedule to a file with the specified name.
	(see Saving schedules)
> *Exit*	exits from the program.
	(see Exit)

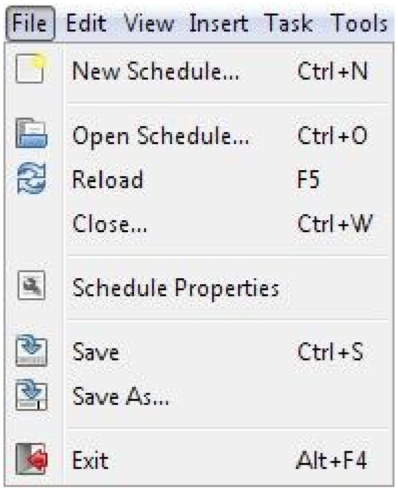


figure 2.2 File menu

Edit

> *Undo*	undoes changes.
	(see Undo and redo changes)
> *Redo*	redoes changes.
	(see Undo and redo changes)

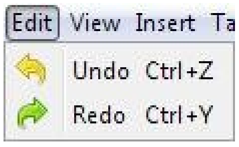


figure 2.3 Edit menu

View

> *Resources*	switches to “Resources” pane. (see “Resources” pane)
> *Tasks*	switches to “Tasks” pane. (see “Tasks” pane)

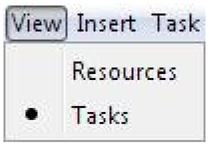


figure 2.4 View menu

Insert

> *New Resource*	for adding a new resource. (see Adding new resources)
> *New Task*	for adding a new task. (see Adding new tasks)

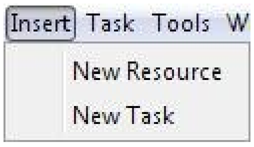


figure 2.5 Insert menu

Task

> *New*	for creating a new task. (see Adding new tasks)
> *Edit*	for editing a focused task. (see Editing tasks)
> *Remove*	removes a focused task. (see Removing of tasks)
> *Edit Predecessors*	for editing predecessors of a focused task.

	(see Editing task's predecessors)
> Edit Resources	for editing resources of a focused task.
	(see Editing task's resources)
> Edit Operations	for editing operations of a focused task.
	(see Editing task's operations)

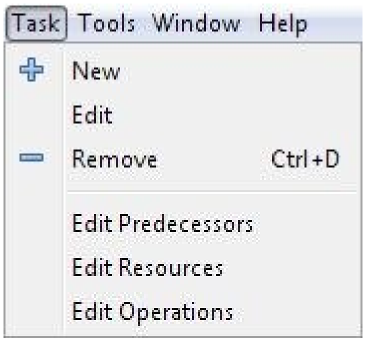


figure 2.6 Task menu

Resource

> New	for creating a new resource.
	(see Adding new resources)
> Edit	for editing a focused resource.
	(see Editing resources)
> Remove	removes a focused resource.
	(see Removing of resources)
> Edit Tasks	for editing tasks of a focused resource.
	(see Editing resource's tasks)

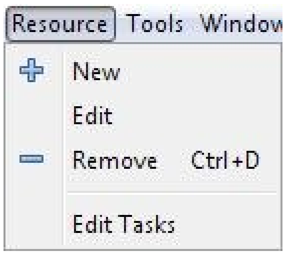


figure 2.7 Resource menu

Tools

> Reschedule reschedules the shown schedule to get a feasible schedule.
(see Rescheduling)

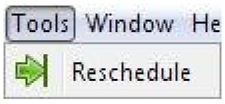


figure 2.8 *Tools menu*

Window

> [an opened file with schedule]

switches to other loaded schedule.

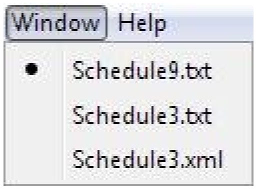
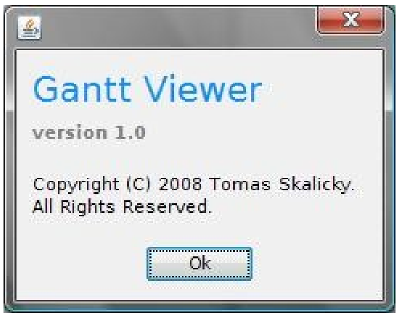


figure 2.9 *Window menu*

Help

> About shows some information about the program.

figure 2.10 *About menu*



Gantt Viewer

version 1.0

Copyright (C) 2008 Tomas Skalicky.
All Rights Reserved.

figure 2.11 dialog About

A.2.2 “Tasks” and “Resources” panes

In these panes, schedules are shown. Both panes are divided into two parts: the left one is a **table** and the right one is a **chart**.

Tables of both panes can be **sorted by any columns** (Task ID, Resource ID - sorted alphabetically; Duration, Start, Finish, Allocated Time - sorted numerically; Predecessors, Resources, Task - sorted by the number of values).

Both charts have **time on x axis**. Imaginary time units are used. The least correct value of time is 0 and the least difference between two values is 1. When the user wants to **zoom** in (out) a chart, he only needs to press the left button of his mouse when the mouse is above a header of a chart and drag his mouse to the right (left). Values in cells of the header change when cells' widths are big or small enough.

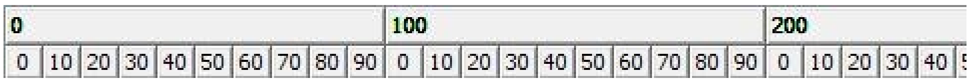


figure 2.12 default view of a header of a chart

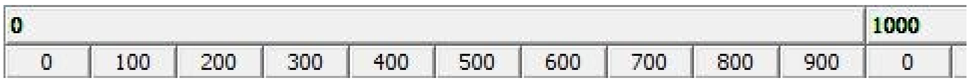
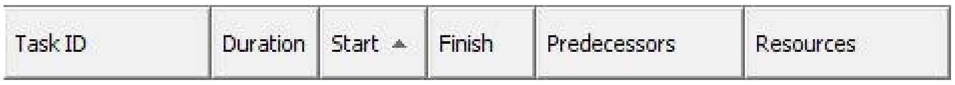


figure 2.13 zoomed out view of a header of a chart

"Tasks" pane

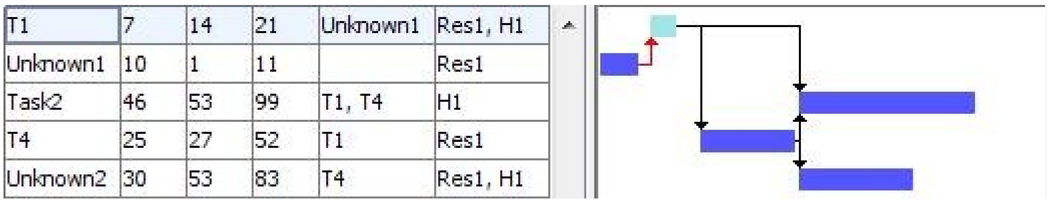
In the table, there are shown all tasks and details about them. All columns are editable.



Task ID	Duration	Start ▲	Finish	Predecessors	Resources

figure 2.14 header of the table in "Tasks" pane

In the chart, there are **images of tasks** and **images of dependencies** between tasks. Each image of a task is next to the row of the table with this task.



T1	7	14	21	Unknown1	Res1, H1
Unknown1	10	1	11		Res1
Task2	46	53	99	T1, T4	H1
T4	25	27	52	T1	Res1
Unknown2	30	53	83	T4	Res1, H1

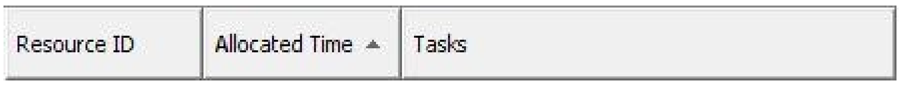
figure 2.15 "Tasks" pane

If the user wants to **move** an image of a task, he has to press the left button of his mouse when the mouse is above the image and drag his mouse where he wants to place the image.
If the user wants to **resize** an image of a task, he has to press the left button of his mouse when the mouse is above the left or the right side of the image and drag his mouse.

(see Editing tasks)

"Resources" pane

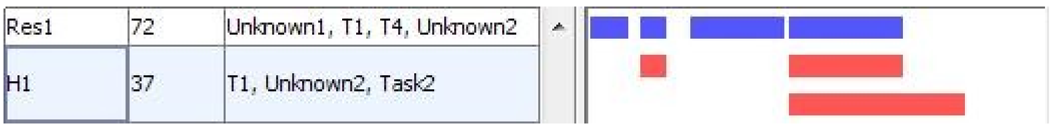
In the table, there are shown all resources and details about them. "Allocated Time" represents how long a resource performs its tasks. All columns except "Allocated Time" are editable.



Resource ID	Allocated Time ▲	Tasks

figure 2.16 header of the table in "Resources" pane

In the chart, there are **images of tasks**. No images of dependencies are displayed there. Each image of a task is next to the row of the table with resource which performs this task. It means that if a task is performed by two resources, there is one image of the task next to the first resource and another image next to the second one.



Res1	72	Unknown1, T1, T4, Unknown2
H1	37	T1, Unknown2, Task2

figure 2.17 "Resources" pane

The user can move and resize images of tasks as in "Tasks" pane.
If the user wants to **move** a task from **one resource to another**, he has to press the left button of his mouse when the mouse is above the image of the task and drag it above the second resource.
(see Editing resources)

Chapter A.3

Creating, opening, closing and saving schedules

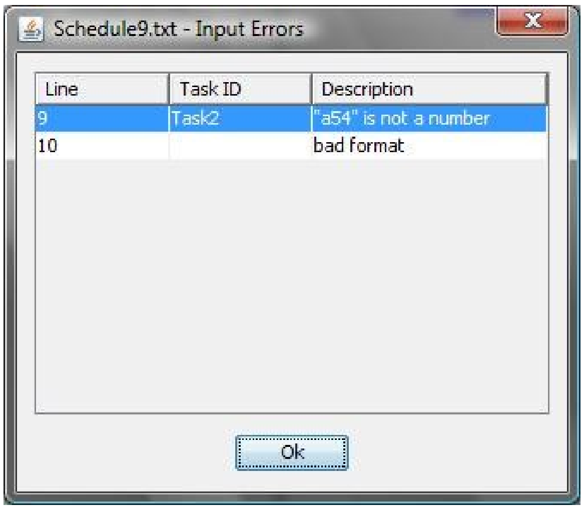
A.3.1 Creating schedules

If the user wants to create a new schedule, he has to click on *File > New Schedule...* or *New Schedule...* in “File toolbar” or type *Ctrl+N*.
The new schedule has an ID in the form “Unknown-[number]” where “[number]” is the first free number of schedules with not specified ID.

A.3.2 Opening schedules

If the user wants to load a schedule from a file, he has to click on *File > Open Schedule...* or *Open Schedule...* in “File toolbar” or type *Ctrl+O*. If the user wants to load the shown schedule again, he has to click on *File > Reload* or *Reload current schedule* in “File toolbar” or type *F5*.

The input file has to be a **text file** and have a fixed format (TXT format or XML format). If the input file is not in a correct format, “Input errors” dialog shows details about **errors** and a schedule which is in that file is loaded without data with errors.



Schedule9.txt - Input Errors

Line	Task ID	Description
9	Task2	"a54" is not a number
10		bad format

figure 3.1 *"Input errors" dialog*

A.3.2.1 TXT format

Lines which begins with '#' character are **comments**. On the first line which is not comment, there is an **ID** of a schedule. Other lines which are not comments represent **tasks**. Each line is divided into cells by tabulators and in these cells, there are *TaskID*, *OrderID*, *ProductID*, *PartsIDs*, *Start*, *Finish*, *PredecessorsIDs*, *ResourcesIDs* in this order.

TaskID	an ID of a task
OrderID	an ID of an order
ProductID	an ID of a product
PartsIDs	IDs of operations which are parts of the task
Start	time when the task starts
Finish	time when the task finishes
PredecessorsIDs . .	IDs of tasks which must finish before the task starts
ResourcesIDs	IDs of resources which perform the task

In *PartsIDs*, *PredecessorsIDs* and *ResourcesIDs*, there are values divided by ',' character.

```
# comment
ScheduleID
TaskID      OrderID      ProductID      PartID,...,PartID      Start      Finish
PredecessorID,...,PredecessorID   ResourceID,...,ResourceID
...
TaskID      OrderID      ProductID      PartID,...,PartID      Start      Finish
PredecessorID,...,PredecessorID   ResourceID,...,ResourceID
```

Definition of TXT format

```
# this is schedule Sch1
Sch1
Task1      O1     P1              10     35              Resource1,Resource2
Task2                     Pt,Pt2  40     50     Task1    Resource1
```

Example of a text file in TXT format

A.3.2.2 XML format

There is a definition of this format in DTD below.

```
<!ELEMENT Schedule (ScheduleID, Tasks)>
<!ELEMENT ScheduleID (#PCDATA)>
<!ELEMENT Tasks (Task*)>
<!ELEMENT Task (TaskID, OrderID, ProductID, Parts, Start, Finish, Predecessors,
Resources)>
```

```
<!ELEMENT TaskID (#PCDATA)>
<!ELEMENT OrderID (#PCDATA)>
<!ELEMENT ProductID (#PCDATA)>
<!ELEMENT Parts (PartID*)>
<!ELEMENT PartID (#PCDATA)>
<!ELEMENT Start (Value)>
<!ELEMENT Finish (Value)>
<!ELEMENT Value (#PCDATA)>                    // integer
<!ELEMENT Predecessors (TaskID*)>
<!ELEMENT Resources (ResourceID*)>
<!ELEMENT ResourceID (#PCDATA)>
```

Definition of XML format in DTD

```
<Schedule>
  <ScheduleID>Sch1</ScheduleID>
  <Tasks>
      <Task>
          <TaskID>Task1</TaskID>
          <OrderID>O1</OrderID>
          <ProductID>P1</ProductID>
          <Parts></Parts>
          <Start><Value>10</Value></Start>
          <Finish><Value>35</Value></Finish>
          <Predecessors></Predecessors>
          <Resources>
              <ResourceID>Resource1</Resource>
              <ResourceID>Resource2</Resource>
          </Resources>
      </Task>
      <Task>
          <TaskID>Task2</TaskID>
          <OrderID></OrderID>
          <ProductID></ProductID>
```

```
            <Parts>
                <PartID>Pt</PartID>
                <PartID>Pt2</PartID>
            </Parts>
            <Start><Value>40</Value></Start>
            <Finish><Value>50</Value></Finish>
            <Predecessors><TaskID>Task1</TaskID></Predecessors>
            <Resources><ResourceID>Resource1</Resource></Resources>
        </Task>
    </Tasks>
</Schedule>
```

Example of a text file in XML format

A.3.3 Closing schedules

If the user wants to close the shown schedule, he has to click on *File* > *Close...* or type *Ctrl*+*W*.
The user is asked whether he wants to save unsaved changes.

A.3.4 Saving schedules

If the user wants to save the shown schedule, he has to click on *File* > *Save* or *Save* in “File toolbar” or type *Ctrl*+*S*. If the user wants to choose a filename and save the shown schedule to a file with it, he has to click on *File* > *Save As...*.
The user can save a schedule in **text file** in TXT format or XML format.

Chapter A.4

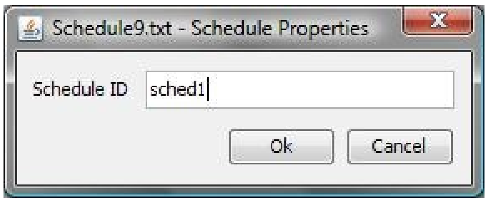
Modifying schedules

A.4.1 Undo and redo changes

If the user adds or removes a task or a resource, he can undo it by clicking on *Edit* > *Undo* or typing *Ctrl+Z*.
The user can also redo changes by clicking on *Edit* > *Redo* or typing *Ctrl+Y*.

A.4.2 Modifying schedule properties

If the user wants to change properties of the shown schedule, he has to click on *File* > *Schedule Properties*. After that, “Schedule Properties” dialog displays.
In the dialog, the user can change an ID of the schedule.



Schedule ID: sched1

figure 4.1 “Schedule Properties” dialog

A.4.3 Adding new tasks

If the user wants to add a new task, he has to click on *Insert* > *New Task*, *Task* > *New* or *New Task* in a popup menu which is above the table of “Tasks” pane or in “Tasks” and “Predecessors” dialog, he can use “New Task” button. After any of these possibilities, dialog “New Task” displays.
In the dialog, the user can edit details of the new task. (see Editing tasks)

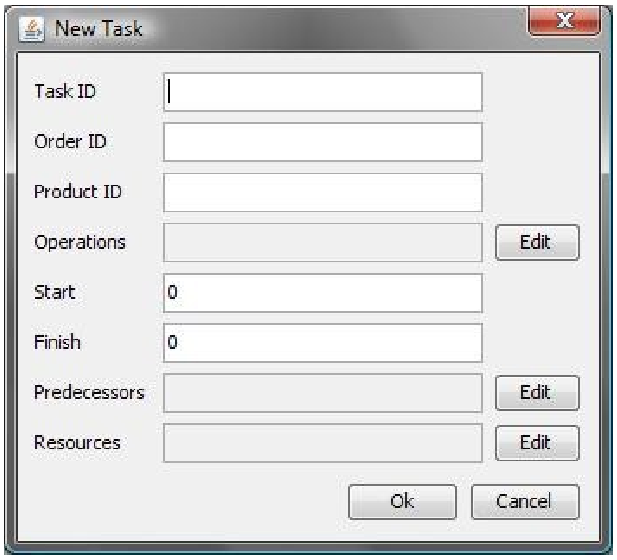
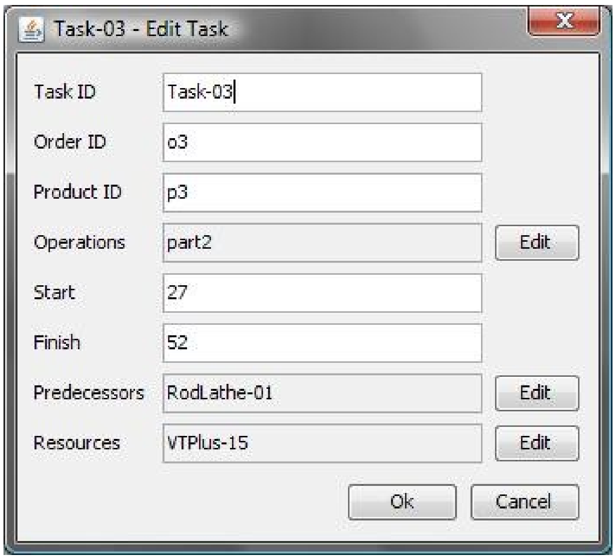


figure 4.2 “New Task” dialog

A.4.4 Editing tasks

If the user wants to edit a focused task, he has to click on *Task* > *Edit* or *Edit Task* in a popup menu which is above the table of “Tasks” pane. After any of these possibilities, dialog “Edit Task” displays.
In the dialog, the user can change details of the focused task.


Task-03 - Edit Task

Task ID	Task-03	
Order ID	o3	
Product ID	p3	
Operations	part2	Edit
Start	27	
Finish	52	
Predecessors	RodLathe-01	Edit
Resources	VTPlus-15	Edit

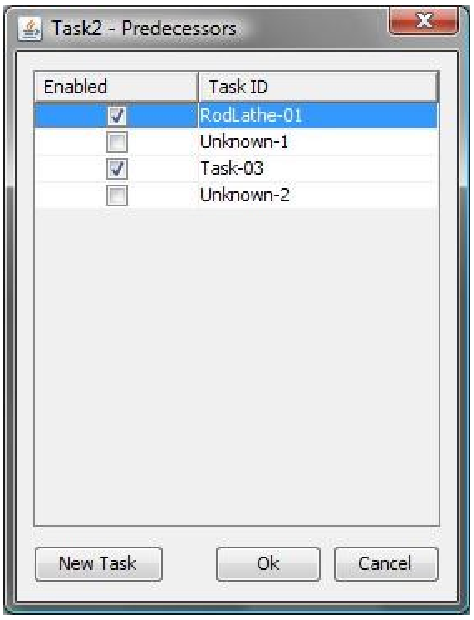
figure 4.3 "Edit Task" dialog

Task ID	an ID of the focused task
Order ID	an ID of an order
Product ID	an ID of a product
Operations	IDs of operations which are parts of the task (see Editing task's operations)
Start	time when the task starts
Finish	time when the task finishes
Predecessors	IDs of tasks which must finish before the task starts (see Editing task's predecessors)
Resources	IDs of resources which perform the task (see Editing task's resources)

If the user only wants to change values of time, he can move or resize an image of the task in any chart. (see "Tasks" pane, "Resources" pane)

Editing task's predecessors

If the user wants to select other predecessors of the focused task, he has to click on *Task* > *Edit Predecessors*. After that, “Predecessors” dialog displays. In the dialog, the user can select tasks, new predecessors of the focused task.

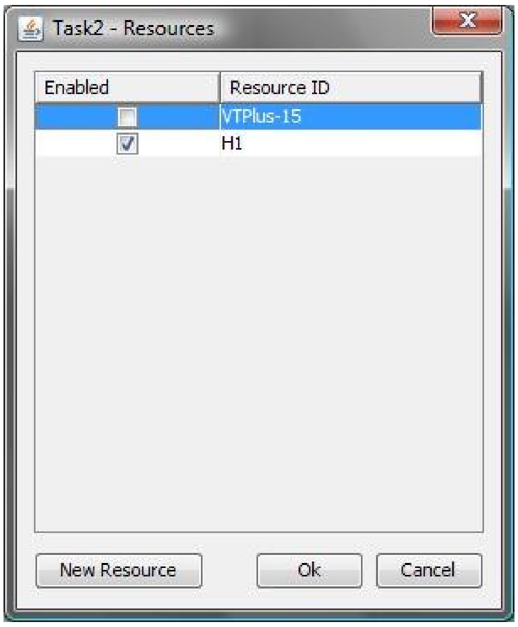


Enabled	Task ID
✓	RodLathe-01
	Unknown-1
✓	Task-03
	Unknown-2

figure 4.4 “Predecessors” dialog

Editing task's resources

If the user wants to change resources of a focused task, he has to click on *Task* > *Edit Resources*. After that, “Resources” dialog displays.
In the dialog, the user can select resources, which perform the focused task.



Enabled	Resource ID
☐	VTPlus-15
✓	H1

figure 4.5 *“Resources” dialog*

Editing task's operations

If the user wants to change operations of a focused task, he has to click on *Task* > *Edit Operations*. After that, “Operations” dialog displays.
In the dialog, the user can modify operations, which are parts of the focused task.

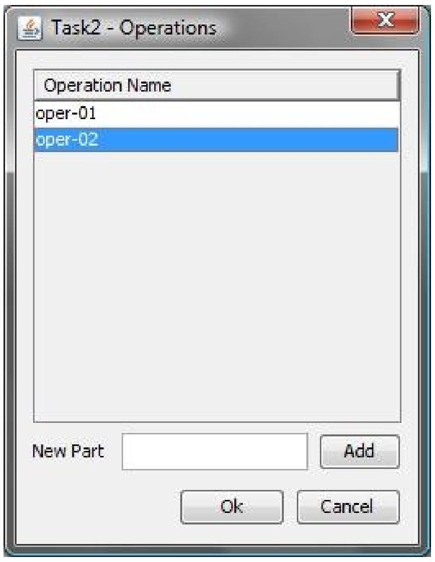


figure 4.6 “Operations” dialog

A.4.5 Removing of tasks

If the user wants to remove a focused task, he has to click on *Task > Remove* or *Remove Task* in a popup menu which is above the table of “Tasks” pane or type *Ctrl+D*.

A.4.6 Adding new resources

If the user wants to add a new resource, he has to click on *Insert > New Resource*, *Resource > New* or *New Resource* in a popup menu which is above the table of “Resources” pane or in “Resources” dialog, he can use “New Resource” button. After any of these possibilities, dialog “New Resource” displays.

In the dialog, the user can edit details of the new resource. (see Editing resources)

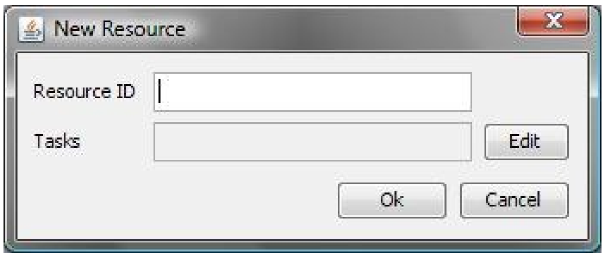
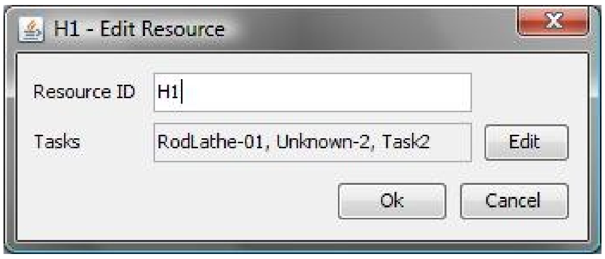


figure 4.7 “New Resource” dialog

A.4.7 Editing resources

If the user wants to edit a focused resource, he has to click on *Resource > Edit* or *Edit Resource* in a popup menu which is above the table of “Resources” pane. After any of these possibilities, dialog “Edit Resource” displays.
In the dialog, the user can change details of the focused resource.



Resource ID: H1

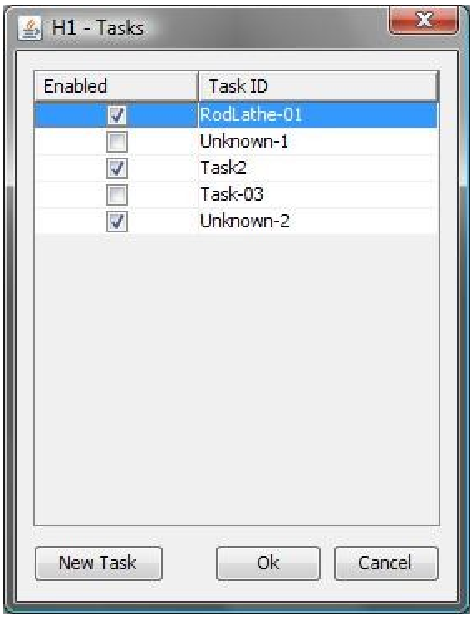
Tasks: RodLathe-01, Unknown-2, Task2

figure 4.8 “Edit Resource” dialog

Resource ID	an ID of the focused resource
Tasks	IDs of tasks which are performed by the resource (see Editing resource's tasks)

Editing resource's tasks

If the user wants to change tasks of a focused resource, he has to click on *Resource > Edit Tasks*. After that, “Tasks” dialog displays.
In the dialog, the user can select tasks, which are performed by the focused resource.



Enabled	Task ID
✓	RodLathe-01
☐	Unknown-1
✓	Task2
☐	Task-03
✓	Unknown-2

figure 4.9 “Tasks” dialog

A.4.8 Removing of resources

If the user wants to remove a focused resource, he has to click on *Resource > Remove* or *Remove Resource* in a popup menu which is above the table of “Resources” pane or type *Ctrl+D*.

Chapter A.5

Rescheduling

Schedule is feasible whether:

1) no task starts earlier than all it's predecessors finish,
2) no resource performs more than one task at a time.

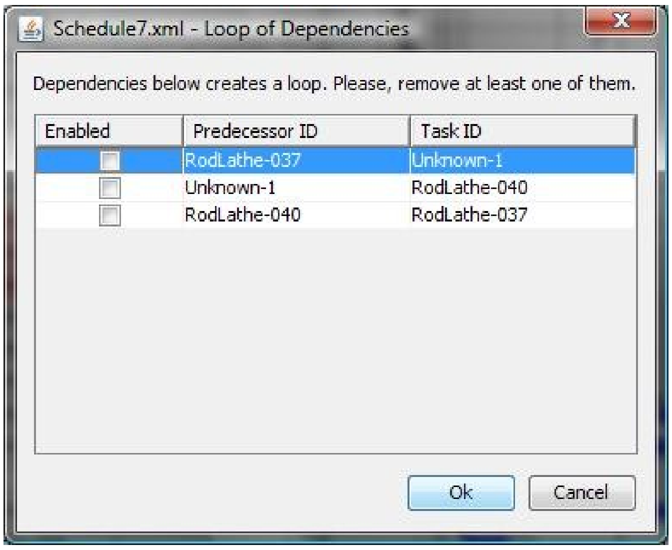
If the user wants to have a feasible schedule, he has to click on *Tools > Reschedule* or *Reschedule* in “Tools” toolbar. Then, rescheduling is performed.
Conflicts are resolved locally. It means that the tasks are shifted as few as possible and nothing is modified except tasks' start time.

There are **two steps** of rescheduling:

1) Removing of cycles of dependencies,
2) Shifting tasks.

A.5.1 Removing of cycles of dependencies

Until there is no loop of dependencies in the schedule, the schedule is checked.
If a loop is found, “Loop of Dependencies” dialog shows dependencies which formed a loop. If the user selects at least one dependence and he clicks on “OK” button, rescheduling continues. If the user selects no dependence and he clicks on “OK” button, the dialog is still visible. If the user selects “Cancel” button, rescheduling is stopped.



Schedule7.xml - Loop of Dependencies

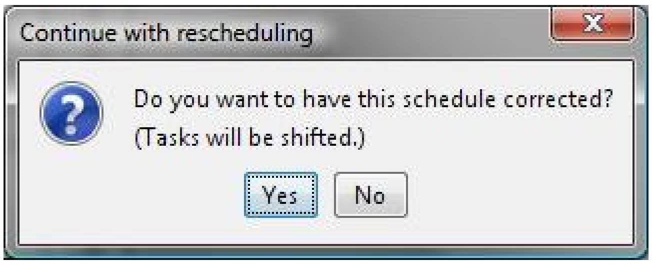
Dependencies below creates a loop. Please, remove at least one of them.

Enabled	Predecessor ID	Task ID
☐	RodLathe-037	Unknown-1
☐	Unknown-1	RodLathe-040
☐	RodLathe-040	RodLathe-037

figure 5.1 “Loop of Dependencies” dialog

A.5.2 Shifting tasks

If there is no loop in the schedule, “Continue with rescheduling” dialog shows. If the user selects “OK” button, tasks are locally rescheduled. If he selects “Cancel” button, rescheduling is stopped.



Continue with rescheduling

Do you want to have this schedule corrected?
(Tasks will be shifted.)

figure 5.2 “Continue with rescheduling” dialog

Appendix B

Contents of CD

On CD which is attached to the bachelor thesis, there are the bachelor thesis itself in PDF format, program GanttViewer using algorithm TmpO-1 and program GanttViewer using algorithm TmpO-2, developer guide, source code of the program and others.

Directory or file	**Contents**
./thesis/interactive_gantt_chart.pdf	this thesis
./thesis/interactive_timetabling.pdf	Master thesis of T. Müller [3]
./gantt_viewer	directory of program GanttViewer
./gantt_viewer/gantt_viewer_tmpo-1.jar	executable file of program GanttViewer which using algorithm TmpO-1
./gantt_viewer/gantt_viewer_tmpo-2.jar	executable file of program GanttViewer which using algorithm TmpO-2
./gantt_viewer/developer_guide.pdf	developer guide of program GanttViewer
./gantt_viewer/user_guide.pdf	user guide of program GanttViewer
./gantt_viewer/installation_guide.pdf	installation guide of program GanttViewer
./gantt_viewer/test_input	directory of test inputs
./gantt_viewer/javadoc.zip	documentation of program GanttViewer generated by JavaDoc
./gantt_viewer/src.zip	source code of program GanttViewer