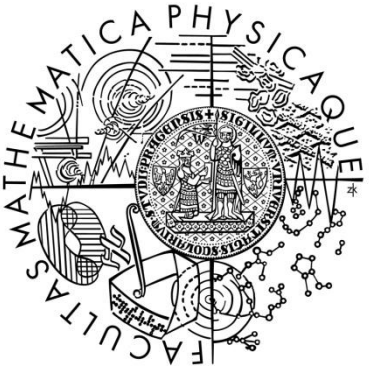
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Tomáš Skalický

Interactive Scheduling and Visualisation

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: doc. RNDr. Roman Barták Ph.D.

Study programme: Informatics

Specialization: Software Systems

Prague 2011

I would like to thank Mr. Roman Barták for supervising this thesis and the FlowOpt project, all members of the FlowOpt team, i.e. Mr. Vladimír Rovenský, Mr. Milan Jaška and Mr. Ladislav Novák, for their persistence and diligence and Mr. Con Sheahan, Mr. Martin Cully and Mr. Dang Thanh-Tung for their help with the FlowOpt project.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…..... date.....

Název práce: Interaktivní rozvrhování a vizualizace

Autor: Tomáš Skalický

Katedra / Ústav: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: doc. RNDr. Roman Barták, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Cílem této diplomové práce bylo navrhnout a implementovat grafický nástroj pro zobrazování a editaci rozvrhů, který by obsahoval funkci pro automatické opravení porušených podmínek v rozvrhu. Výsledná aplikace Gantt Viewer je integrována do projektu FlowOpt, jenž představuje komplexní řešení od modelování pracovních procesů, přes vytváření rozvrhů, až po jejich analýzu. Aplikace byla vyvíjena s důrazem na přívětivé uživatelské rozhraní a na výkonnost při práci s velkými daty. Díky integraci do FlowOpt projektu umožňuje zobrazit u rozvrhů informace o pracovních procesech. Gantt Viewer též obsahuje nástroj pro automatické opravení rozvrhů využívající Repair-DTP algoritmus, který je v této práci podrobně představen a demonstrován.

Klíčová slova: Rozvrhy, FlowOpt, Přerozvržení, Pracovní postupy, DTP

Title: Interactive Scheduling and Visualization

Author: Tomáš Skalický

Department / Institute: Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: doc. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: The goal of this thesis was to design and implement a graphical tool for visualization and editing of schedules which would provide a function for automatic repairing of violated constraints in the schedule. The resulting application called a Gantt Viewer is integrated to the FlowOpt project that represents a complex solution for modeling workflows, creation of schedules from them and analysis of these schedules. The application has been developed with the focus on intuitiveness of the user interface and performance during the management of large schedules. It enables the user to visualize extended manufacturing schedules thanks to the cooperation with other modules of the FlowOpt project. Moreover, the Gantt Viewer incorporates a repair tool exploiting a new Repair-DTP algorithm which is both introduced and demonstrated in this work.

Keywords: Schedules, FlowOpt, Rescheduling, Workflows, DTP

Contents

Introduction

Schedules are used in many areas of human activities and among the most popular are public transportation and project management. These areas have their specifics which are reflected in the schedules, but still, their schedules have one in common; all of them consist of three concepts: activities, resources and constraints. Activities are jobs which need to be performed, resources are designated for activities' execution and constraints introduce various limitations among the activities as well as the resources.

This thesis concerns with one type of schedules which is used in factories and help with manufacturing of various products, e.g. manufacturing of a car. For such schedules it is characteristic that there exist patterns for all products which need to be produced. These patterns, usually called workflows, determine how to manufacture the products. Therefore, if two products of the same type are to be manufactured, they share the same workflow.

In the first part of this work we present a graphical tool for visualization and editing of the manufacturing schedules. The tool provides the user with a user-friendly interface and allows him to work with large data. Nevertheless, the main added value of the tool in comparison to a vast majority of current applications is the utilization of knowledge about the workflows associated with the schedules. This knowledge is available thanks to the integration of the tool into a system where the workflow editor is also part of, which allows us to visualize more aspects of the schedules.

The second part of this thesis deals with the problem of automatic correction of flaws in the manufacturing schedules. There are two commonly known approaches to solve this problem. The first one takes activities, resources and constraints from the original infeasible schedule and conducts scheduling of them from scratch. The second approach is wholly different. It takes an infeasible schedule and makes such modifications that the resulting schedule is feasible.

The algorithm, which we suggest, comes under both of above groups. Its goal is to return the user such a schedule which is feasible and is as similar to the original schedule as possible. The algorithm works basically in two stages: first it constructs a feasible schedule from scratch (the first approach). For this purpose, it exploits a network of constraints which provides us with their propagation [6]. Afterwards, in the second stage, the algorithm takes the original infeasible schedule and starts to apply such operations that correct the existing violations (the second approach). The basic idea of these corrections is that:

- the intermediate schedule N (schedule after n corrections) is less or equally similar to the original infeasible schedule as the intermediate schedule M (schedule after $m < n$ corrections) and
- the intermediate schedule N is more or equally similar to the solution of the first stage as the intermediate schedule M.

Therefore, if no better solution is found, the solution of the first stage is returned. We should add that the only operation that we consider throughout the whole algorithm is a shifting of activities in time; their durations and resource allocations are fixed.

The thesis has nine chapters. The first three chapters introduce the background of the whole work and an exact description of manufacturing schedules which are considered there.

The chapters 4 and 5 discuss the first aim of this thesis – creation of an application for visualization and editing the manufacturing schedules. We familiarize there with the application Gantt Viewer and with its user interface.

The chapter 6 contains a discussion about the second goal of this work – a proposition of a repair algorithm. We define there the problem of fixing violations which we are solving precisely. Then, the following chapters go first through the existing approaches which solve this problem or similar ones and afterwards, a new repair algorithm called Repair-DTP is presented together with the proof of algorithm's completeness and soundness.

The last part of this work is devoted to experiments with the proposed repair algorithm. They demonstrate how the algorithm behaves in practice.

1. Gantt Viewer, FlowOpt and MAK€

In this chapter we will familiarize with the background of the Gantt Viewer and with the consequences which follow from it. They influence the visualization of the schedules in the viewer as well as the efficiency of the repair algorithm introduced later on.

The Gantt Viewer was developed as a part of the FlowOpt software project. The goal of the project was to provide the user with an application which helps him to design *manufacturing workflows* and to obtain an appropriate optimized[1] *schedule* useful in practice (e.g. in factories). The key features of the project were an intuitive user-friendly interface and getting the schedule in a short time.

Manufacturing workflows are descriptions of specific (business) processes. They show what activities need to be processed to get a particular product, which of them is/are first and which come after, what resources can be allocated to the activities and so forth. However, the time allocation of activities is not decided since the workflow is only a recipe "How to manufacture a …"

Manufacturing of particular product(s) is depicted by a schedule. The schedule tells the user when each activity is planned, which resources will really perform an activity, when the product(s) is/are finished etc. In short the schedule works with instances of the workflows.

The intermediate structure between the workflows and the schedule is *a work order*. It contains information about how many instances of each workflow the schedule will contain, when the manufacturing will start and when it should end – *due date*.

The FlowOpt project consists of five modules: Workflow Editor, Work Order Manager, Scheduler, Gantt Viewer and Schedule Analyzer. There is a short scenario of using all of them:

1. First of all, the user designs *workflows*, which he is interested in, in the Workflow Editor. Let us assume that the user wants to produce tables and chairs. It follows that he has to describe two workflows; one for a table and the other for a chair. For each activity of these workflows it is necessary to say how much time it takes, which resources can carry it out and so on. The

[1] The resulting schedule of the FlowOpt project is optimal if the FlowOpt's scheduling engine has enough time. The reason is that the scheduling problem in the FlowOpt project is NP-hard. As it will turn out in the chapter Conversion to DTP, we can represent the FlowOpt's schedule as a DTP [14]. Since we can also do the opposite conversion (e.g. done during the returning of the resulting schedule of the algorithm suggested in this thesis) and it is proved that the DTP problem is NP-complete [10], the scheduling is NP-complete as well.

user can even specify alternatives, e.g. he can either manufacture a table on his own, or buy it from a contractor.

2. The following step is to determine how many units of each item the user wants to manufacture. For that purpose he should use the Work Order Manager which assists him with the creation of a *work order*.
3. Afterwards, the Scheduler creates a *schedule* from appropriate workflows according to the work order. It means that the Scheduler assigns exact temporal data to all activities, selects resources for each activity which execute it, selects the most suitable alternative among the available ones and so on. All these actions are done with respect to a cost function which enables the Scheduler to compare two schedules. It tries to find an optimal feasible schedule. However, since the problem is NP-hard, it is likely that for big workflows or large orders the Scheduler does not find an optimal solution in a short time. Therefore, the user should specify how long the engine can run. If it reaches this limit, the Scheduler returns the best feasible schedule it has found till that time.
4. Now, the user can visualize the obtained schedule in the Gantt Viewer which provides him with two possible views: a Gantt Chart and a Resource View (see Visualization and Interactivity). He can also make some operations here, but the schedule still has to be consistent with the workflows from which it has been created.
5. The last of the modules analyzes a feasible schedule and proposes actions which the user (factory) should do in order to improve efficiency. Such an action can be for instance buying a new machine.

Since the Gantt Viewer works with the schedules which are based on the workflows designed in the Workflow Editor, data related to these workflows are propagated further to the Gantt Viewer. Although these data are not vitally important for visualization and the repair algorithm, they are useful. In comparison with the applications which do not consider workflows, the Gantt Viewer can visualize more, e.g. alternatives which have not been selected by the Scheduler. Later on, we will demonstrate that information about the tree structure of the workflow can improve efficiency of the repair algorithm significantly (see Conversion of Constraints).

The FlowOpt project was developed in cooperation with ManOPT Systems Ltd and that led to an integration of the project to the MAK€ application developed by this company. The MAK€ application was robust and treated the same problem as FlowOpt, but with more functions and extensions necessary in practice.

The integration was useful not only from the programmer's perspective, but also from a theoretical and academic one. References to useful third-party libraries, knowledge of a domain of workflows and scheduling and feedbacks based on experiences gained from business world were great assets to the Gantt Viewer.

If you want to find out more information about the FlowOpt software project, feel free to go through its documentation provided on the attached CD-ROM.

2. Gantt Viewer's Schedules

In the previous chapter we have already loosely described what the schedules are. Now we focus on their complete introduction and on the process how they are created. For the formal definitions, see Formal Definition of the Model.

2.1 FlowOpt's Workflows

First it is essential to say how the workflows in the Workflow Editor look like. It has a crucial impact on the representation of schedules in other FlowOpt modules, including the Gantt Viewer. Anyway, the description which follows is not complete; it considers only those parts which are important for the schedules in the Gantt Viewer. For instance the preferred routes are missing there. For the complete description of the workflows, see the user documentation of the FlowOpt project, or [11].

2.1.1 Nested TNA

Before we introduce the workflow's structure, we need to familiarize with a *Nested TNA* [2]. We will see later on that the former is an extension of the latter.

A Nested Temporal Networks with Alternatives (TNA) is a tree-like structure which is useful for a representation of for instance a more complicated work which can be divided into smaller tasks. The inner nodes of the structure represent *nests* which decompose further into one or more other nodes. The leaves of the structure are nodes annotated by two temporal variables – one variable holds a start time of a particular leaf and the other variable holds its end time.

There are three possible decompositions in the Nested TNA structure: *serial, parallel* and *alternative*. The nest with the serial type decomposes into a sequence of one or more nodes. The parallel decomposition, as the name suggests, is a kind of complement to the serial type. There is no order among the nodes and they can be processed in parallel. The alternative decomposition is a concept of a different kind. It introduces to the workflow's structure a possibility to choose. Exactly one of the children of an alternative nest can be eventually executed.

We should say that the preceding description of the Nested TNA structure is not exactly the same as in [2]; rather it is a loose interpretation. In [2], there are no decompositions of nests, but constraints which are among the nodes. However, this difference has no impact on the correctness of the description presented above.

An example of the Nested TNA structure follows in Figure 1:

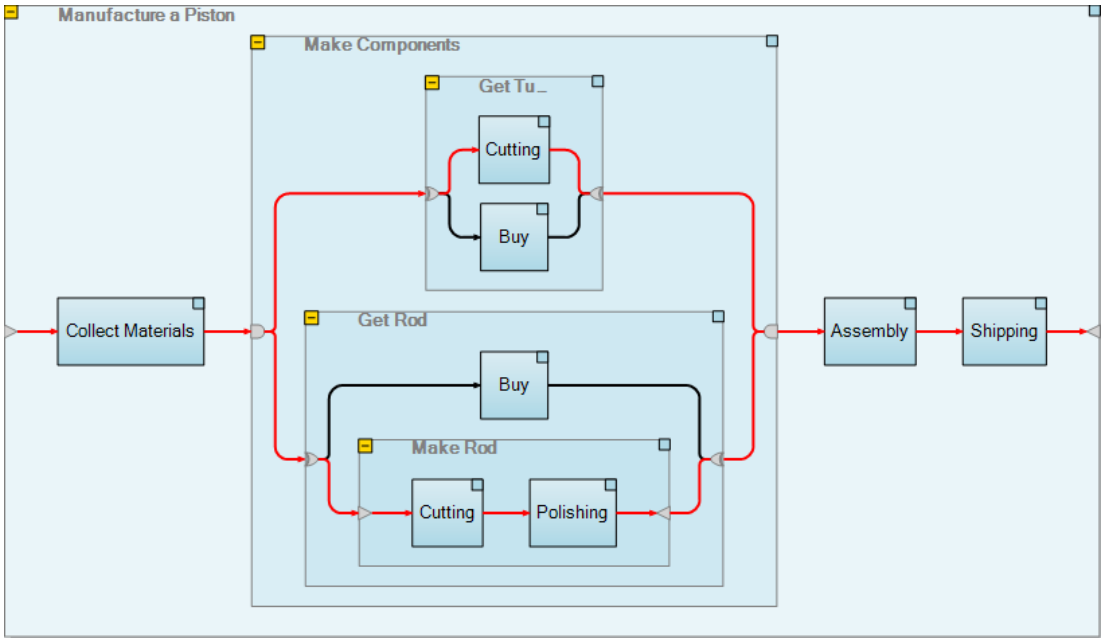


Figure 1: The Nested TNA structure for manufacturing of a piston

The figure represents a manufacturing of a piston. The root nest "Manufacture a Piston" decomposes into four other nodes: "Collect Materials", "Make Components", "Assembly" and "Shipping". Since the decomposition is serial it means that the first thing which is to be done is a collection of necessary materials and the last thing is a shipping of a new piston to a customer. The order cannot be changed. Except the second child, i.e. "Make Components", all children of the root nest are leaves. The child "Make Components" is slightly complicated; it decomposes into two nodes which are to be executed in parallel: "Get Tube" and "Get Rod". It means that we can get a tube and a rod of the new piston independently on each other. Both "Get Tube" and "Get Rod" are nests with the alternative decomposition. We can recognize that by following the black links since the red ones highlight the preferred routes, i.e. preferred alternatives (for more details see [11]).

2.1.2 Workflow

Workflow is an extension of the Nested TNA structure. So, the workflow is also a tree-like structure where we will call all nests *tasks* and all leaves will be *activities*. Together they create a group of *nodes*.

2.1.3 Nodes

Tasks and activities differ a lot, but still they have a few features in common. Each pair of nodes can be connected with a constraint (see Constraints).

The main differences between the tasks and the activities are:

- Location – An activity is a leaf of the workflow, a task is an inner node.
- Input of temporal data – Temporal data of an activity are assigned explicitly; task's temporal data are computed.

- Allocation of resources – An activity requires resources for its execution, a task does not. Precisely, an activity can require resources, but it is possible to introduce such activities which do not need any resource.

Selected nodes

Before we continue with the description of the workflow's structure, we should introduce a predicate *selected*. The term *selected* is used in this thesis in two meanings: one refers to the nodes and the other to the resource groups (reservations). Here, we will concern only the first one (for the second one see Selected resource groups). Since the workflow's structure is an extension of the Nested TNA, the concept of an alternative decomposition is available there as well. The concept is useful when we want to express that there are more possibilities how to reach a certain goal (= how to execute a certain task) and these possibilities are disjunctive. Thus, if we add a task with the alternative decomposition into our workflow, the decomposition secures that if the task is *selected*, exactly one task's child is *selected* and all the others are not.

In the rest of this work, we will call the schedule created by all selected nodes either *a resulting schedule*, or *a solution*. Although these terms can also appear in different meanings, the proper meaning will be always clear from the given context. Moreover, instead of *the selected node* (activity, task) we will sometimes use *the scheduled node* (activity, task). Their meanings are same.

Tasks

The inner nodes of a workflow are called tasks and since they are inner, they have always at least one child node. One particular task is fulfilled (= finished) if and only if the task is selected and all of its selected children are fulfilled. In other words, execution of the task starts when its first child starts and ends when its last child ends. The difference between these two points of time gives us the duration of the task.

There are three types of tasks: serial, parallel and alternative. It is apparent that they refer to the types of decompositions in the Nested TNA structure.

- The children of a selected *serial task* are performed sequentially. It follows that the first and the last children can be unambiguously identified.
- *A parallel task* is a counterpart of the serial one. There is no order among the children of a selected parallel task and it implies that it is not possible to say which child would be the first in the resulting schedule and which would be the last.
- *An alternative task* enables the user to define alternatives. We will sometimes use a term *branches* instead of alternatives and children of an alternative task. For each selected alternative task the Scheduler should select exactly one of its children to the resulting schedule, neither more, nor less.

Note that we said nothing about the not-selected serial (parallel, alternative) tasks. These tasks introduce no constraints to the resulting schedule, but still we count them among the serial (resp. parallel, alternative) tasks.

Activities

The leaves of the workflow tree are called activities. Activity's start and end times are not defined in the Workflow Editor. These temporal data are determined by the Scheduler and they can be modified in the Gantt Viewer.

Anyway, it is necessary to set up an execution time of an activity, i.e. *duration*, in the Workflow Editor. This time is not associated with the activity directly; it is defined via the resources which can potentially carry out this activity. The user can even set different activity's durations for different resources. The conversion of such a case from the representation in the workflows to one in the schedules is not so straightforward; for more details, see Instantiation of workflows.

We should emphasize that an activity can require more resources for its execution; we would rather say *resource groups*. Each of these groups can contain real resources. It can be illustrated on an example of activity "Cutting tree". For its execution a man and a saw is necessary. It follows that there are two groups of resources; one of workers which can saw and one of available saws. It is up to the Scheduler or the user in the Gantt Viewer which man and which saw will be really selected.

Selected resource groups

Resource groups are the second case when the predicate *selected* is used. It says that if an activity is selected, exactly one resource in each resource group associated with this activity is *selected*. The *selected* resources from all these groups then perform the activity.

2.1.4 Resources

The subject which can perform an activity is called a resource. It can be for instance a worker or a machine. All resources have one in common; they can carry out at most one activity at a time – *unary resources*. Let us call this limitation *a resource constraint*.

In the FlowOpt project we consider that each resource is available all the time.

2.1.5 Constraints

Constraint is a relation between two nodes which restricts either their mutual position, or their mutual occurrence in the resulting schedule. There are three basic types of constraints: precedence, synchronization and logical constraint.

- The first one – *precedence* – says that if both associated nodes are selected, one has to end before the other starts.

- As the name indicates, *synchronization* makes two points of time of two selected nodes synchronous. There are four types of it:
    - *start-to-start* (SS) – both nodes have to start at the same time.
    - *start-to-end* (SE) – the first node has to start exactly at the same time as the other ends.
    - *end-to-start* (ES) – the first node has to end exactly at the same time as the other starts. Thus, this type of synchronization is a special case of the precedence constraint.
    - *end-to-end* (EE) – both nodes have to end at the same time.
- The last type of constraints is different from the others. Precedencies and synchronizations restrict the mutual positions of associated nodes if the nodes are selected. *The logical constraints* limit mutual occurrences of associated nodes in the resulting schedule. There are three types of these constraints:
    - Implication – if the first node is performed (= selected), the other has to be as well.
    - Equivalence – both nodes have to be either performed, or not performed.
    - Mutual exclusion (MUTEX) – either the first node or the second one can be performed, but not both of them.

Note that there are *implicit constraints* introduced to a workflow by using the tasks. For instance a serial task connects its children with the precedencies. Moreover, the first child is connected with its parent with the SS synchronization; similarly for the last child. The constraints which are added to the workflow explicitly by the user are called *custom constraints*.

2.1.6 Example

We can see that the workflows in the Workflow Editor are quite straightforward and not so complicated structures. The Nested TNA structure has certainly a positive effect on that. Another reason is that there are only four concepts: tasks, activities, constraints and resources; and only the first three of them are visible in the workflow diagrams in the Workflow Editor. We can see that in the following Figure 2.

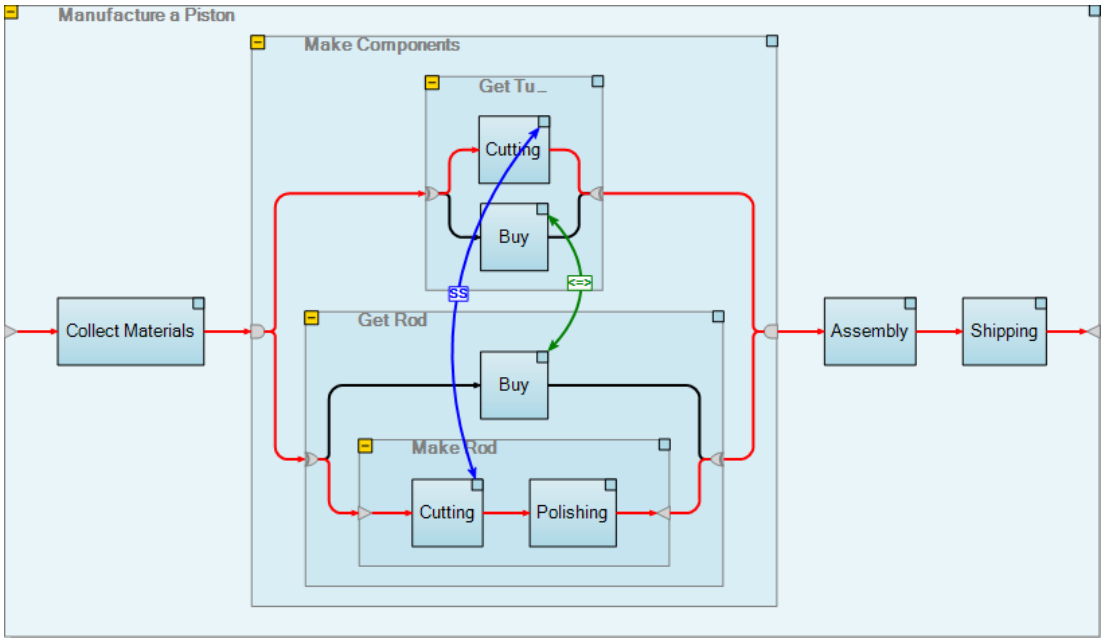


Figure 2: Workflow for manufacturing of a piston

The figure contains almost the same manufacturing process as Figure 1 since as it was said, the workflow's structure extends the Nested TNA one. The only differences between the figures are two custom constraints – one SS synchronization and one equivalence. The meaning of the former is intuitive. The latter one says that a tube and rod are either both bought, or both manufactured, but it is not allowed that one of them is bought and the other is not.

2.2 Going from Workflow to Schedule

Till now we have familiarized with the workflows and with their structure. The following text is about the process how the workflows are converted to schedules common for both the Gantt Viewer and the Schedule Analyzer. Specifics of Gantt Viewer's Schedules are described in the following chapter.

There are two phases of the conversion:

1. Instantiation of the workflows and
2. Scheduling.

The first phase needs to be done before the second one.

2.2.1 Instantiation of workflows

This phase is about adding as many instances of the workflows to a schedule as it is specified in the work order, i.e. if two pistons (see Figure 2) are required, there are two root tasks "Manufacture a Piston" in the schedule. It means that the schedule is a forest in the graph terminology and each of its trees (root nodes) has the same tree-like structure as the corresponding workflow.

However, there is one exception. The reason is a restriction in the visualization of schedules in the Gantt Viewer (see Swapping of Reservations). The exception concerns the activities in the workflow that have more resources which can perform them and of which execution times differ. Such activities are converted to alternative tasks. These tasks have as many children as there are different execution times; each of the children is a copy of the original activity and is associated just with the resources which have the same execution time. The simple example is depicted by the following figures:

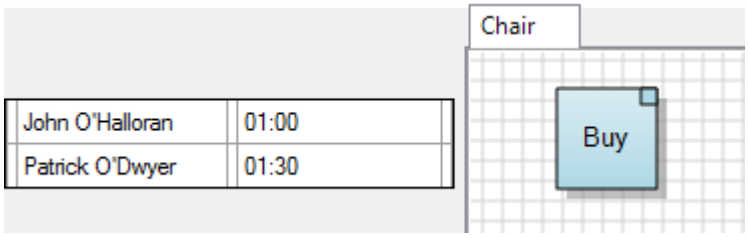


Figure 3: Workflow of manufacturing (buying) of a chair

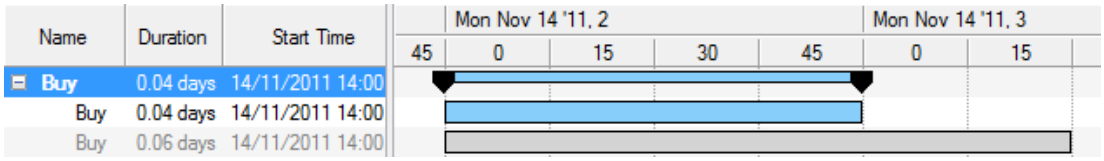


Figure 4: Schedule with one chair manufactured according to the workflow in Figure 3.

In Figure 3 we can see a workflow of manufacturing a chair. The chair is not manufactured, actually it is bought and there are two men which can do that. The first one, John O'Halloran, manages buying one chair in one hour. The second man, Patrick O'Dwyer, does that in one hour and half. Although the whole workflow consists of just one activity "Buy", it is converted to an appropriate schedule as an alternative task "Buy" with two alternatives (see Figure 4). Note that the Scheduler decided that John will buy the chair.

2.2.2 Scheduling

The scheduling phase follows immediately after the instances of the workflows are created. Its goal is to organize all nodes of the schedule in a way that the schedule would be *consistent* and *optimal*.

Consistency

The schedule is consistent when all of the following conditions are satisfied:

- All constraints (precedencies, synchronization, logical and resource) are satisfied.
- Exactly one child of each selected alternative task is selected.
- Exactly one resource in each resource group is selected if an associated activity is selected.

Optimality

Schedule is optimal when it satisfies a due date specified in a work order the best it can. The phrase "satisfies a due date" means that the last selected activity in the schedule ends not later than the due date comes. Since the problem of finding the optimal schedule is NP-hard, it may take a fair amount of time to get such a schedule. The Scheduler hence rates each solution which it finds and if the solution is better than the best which the Scheduler has found till now, the Scheduler reports that to the user (see Figure 5). The user can stop the scheduling whenever he wants and he can store the best solution (if any exists) to the database. It follows that the user starts working only with a consistent schedule in the Gantt Viewer.

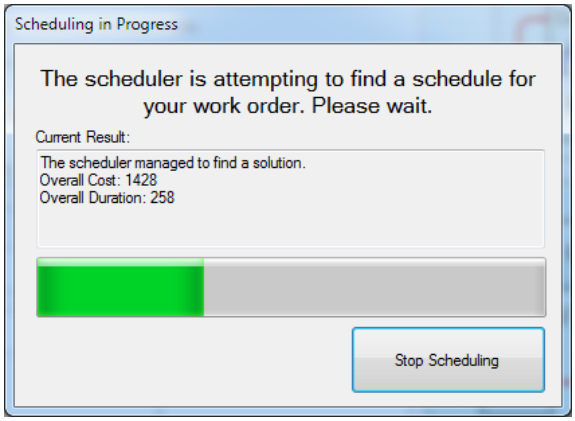


Figure 5: The best schedule which the Scheduler has found till now has rating 1428 (cost) / 258 (duration).

Possible operations

The scheduling engine can use tree types of operations to find a consistent and optimal schedule:

- The engine can assign an arbitrary value to the start time of a selected activity. Other temporal data cannot be changed by the Scheduler. Thanks to the normalization of resources done during the Instantiation of workflows, we can consider that each selected activity in the resulting schedule has one value of duration determined in the Workflow Editor. The values of the end times are redundant since they equal sums of the start times and the durations. Furthermore, temporal data of all tasks are computed from their children.
- The engine can choose which branches in selected alternative tasks will be selected.
- The engine can choose which resource in each resource group associated with the selected activity will be selected.

Note that the Scheduler must set up the start times of all selected activities, select exactly one branch for each selected alternative task and select exactly one resource from each resource group associated with any selected activity to get a feasible solution with which we can work in the Gantt Viewer.

2.3 Specifics of Gantt Viewer's Schedules

In the previous chapters we have found out how the workflows look like and how the schedule is built from them. Let us describe now the extensions of the schedules that the Gantt Viewer introduces to them.

There are three operations which have been added:

- Banning of branches,
- Banning of resources,
- Pinning of activities.

The first two operations allow the user to ban either a not-selected branch, or to ban a resource which is allocated to no activity. If the branch is banned, the user cannot schedule it (= make it selected). If the resource is banned, it cannot be allocated by any activity.

Pinning is likely more important extension than banning. The pinning enables the user to place pins upon scheduled activities. As a result, pinned activities cannot be moved during the run of the repair algorithm suggested in this thesis. All the other activities can be shifted anywhere.

Since banning and pinning are advanced operations, we will go through them in detail later on in the chapters Banning and Pinning.

3. Formal Definition of the Model

In the previous chapter we familiarized with the process how the schedule is created and with the schedule's structure. In the following text we will describe the Gantt Viewer's schedules formally and we will list all implicit and custom constraints which exist in the schedules.

We should say that some parts of this chapter has been borrowed from the chapter 5.2 in [11] and adjusted to terms introduced in this thesis. The reason is a similarity of the FlowOpt's workflows and schedules which the chapter Going from Workflow to Schedule is talking about.

3.1 Definition of Model

This chapter focuses on the definition of the model of Gantt Viewer's schedules. For that purpose, we will use terms which we have already introduced in the chapter Gantt Viewer's Schedules. In the first part of the definition, we will go through the concepts common for the workflows and the schedules. The second part extends the definition with schedules-specific concepts.

3.1.1 Concepts Common for Workflows and Schedules

Gantt Viewer's schedule *S* is a triple *(Nodes, Resources, Constraints):*

- $Nodes = \{all\ nodes\ in\ S\}$
- $Resources = \{all\ available\ resources\ in\ S\}$
- $Constraints = \{all\ custom\ constraints\ in\ S\}$

We can continue with the introduction of notions related to the nodes. Nodes are organized in a forest structure. It follows that each node *N* either has a parent, i.e. $\exists P \in Nodes: Parent(N) = P$, or is a root. If the node *N* is a root, *Parent(N) = nil*.

- $Children(N) = \{C \in Nodes | Parent(C) = N\}$
- $Ancestors(N) = \begin{cases} \{P\} \cup Ancestors(P) & if\ P = Parent(N) \neq nil \\ \emptyset & otherwise \end{cases}$
- $Roots = \{R \in Nodes | Parent(R) = nil\}$
- $Activities = \{A \in Nodes | Children(A) = \emptyset\}$
- $Tasks = \{T \in Nodes | Children(T) \neq \emptyset\}$

Note that the set *Ancestors* of the root node is empty.

There are three types of tasks: serial, parallel and alternative tasks.

- $Tasks = SerialTasks \cup ParallelTasks \cup AlternativeTasks$
- Each two sets of *SerialTasks*, *ParallelTasks* and *AlternativeTasks* are mutually exclusive.

Now, we focus on the custom constraints. Constraint *C* is a relation between two nodes *M* and *N* which is oriented. Then, we can write the constraint *C* like an ordered pair of nodes *M* and *N*:

$$(\boldsymbol{M}, \boldsymbol{N}) \in \boldsymbol{Constraints}$$

There are three basic groups of the custom constraints: precedence constraints, synchronizations and logical constraints:

- $Constraints = Precedencies \cup Synchronizations \cup Logical$
- $Precedencies = \{C \in Constraints | C\ is\ a\ precedence\}$
- $Synchronizations = SS \cup SE \cup ES \cup EE$
- $Logical = Implications \cup Equivalences \cup MUTEXes$
- Each two sets of *Precedencies*, *SS*, *SE*, *ES*, *EE*, *Implications*, *Equivalencies* and *MUTEXes* are mutually exclusive.

The semantics of all types of the custom constraints will be explained later in the chapter Schedules-specific Concepts.

The final part of the definition common for both the workflows and the schedules is about reservations and resource groups. *A reservation* is a resource *R* which is associated with a particular activity *A* and can perform any of the *A*'s parts, i.e. $R \in CanPerform(A)$. For instance an activity "Cutting Tree" needs one man and one saw for its execution. The activity is associated with men Jerry and Tom and saws Bosch and Goodluck. Then, reservations of the activity are all these four resources.

A resource group G of the activity *A* is a subset of reservations which can perform exactly the same part of *A*, i.e. $\forall P, R \in G: CanPerformPart(A, P) = CanPerformPart(A, R)$. The activity "Cutting Tree" mentioned above has two resource groups: one contains all men and the other all saws.

- $Reservations(A) = \begin{cases} \{R \in Resources | R \in CanPerform(A)\} & if\ A \in Activities \\ \emptyset & otherwise \end{cases}$
- $ResourceGroups(A) = \left\{ G \subseteq Reservations(A) \middle| \begin{matrix} \forall P, R \in G: \\ CanPerformPart(A, P) = CanPerformPart(A, R) \end{matrix} \right\}$

3.1.2 Schedules-specific Concepts

This section is characteristic with usage of the predicates *selected* which we familiarized with in the chapters Selected nodes and Selected resource groups. The selected node is a node which belongs to the resulting schedule.

- $SelectedNodes \subseteq Nodes$
- $SelectedActivities = Activities \cap SelectedNodes$
- $SelectedChildren(N) = Children(N) \cap SelectedNodes$

Each $N \in Nodes$ has temporal data which are determined if and only if $N \in SelectedNodes$. Otherwise, the data are not set. The data consist of three items:

- *Start(N)* is a point of time when an execution of *N* starts.
- *Duration(N)* says how long *N* is executed.
- *End(N) = Start(N) + Duration(N)*

Then, we can define functions:

- $Start, Duration, End: SelectedNodes \rightarrow \mathbb{R} \geq 0$
- $TemporalData: SelectedNodes \rightarrow \mathbb{R}^2 \geq 0$

where the results of the function *TemporalData* are tuples *(Start(N), End(N))* and *N* is a selected node.

If $N \in Activities$, *Start(N)* and *Duration(N)* are determined explicitly (see Possible operations). If $N \in Tasks$, the functions *Start* and *End* satisfy the following conditions:

$$Start(N) = Min_{C \in SelectedChildren(N)}\big(Start(C)\big)$$

$$End(N) = Max_{C \in SelectedChildren(N)}\big(End(C)\big)$$

There are eight different types of custom constraints of which semantics we want to introduce now. We are starting with the precedencies and the synchronizations:

- $(M, N) \in Precedencies$:
$$M, N \in SelectedNodes \Rightarrow End(M) \leq Start(N)$$
- $(M, N) \in SS$:
$$M, N \in SelectedNodes \Rightarrow Start(M) = Start(N)$$
- $(M, N) \in SE$:
$$M, N \in SelectedNodes \Rightarrow Start(M) = End(N)$$

- $(M, N) \in ES$:
$$M, N \in SelectedNodes \Rightarrow End(M) = Start(N)$$
- $(M, N) \in EE$:
$$M, N \in SelectedNodes \Rightarrow End(M) = End(N)$$

Description of the logical constraints follows:

- $(M, N) \in Implications: M \in SelectedNodes \Rightarrow N \in SelectedNodes$
- $(M, N) \in Equivalences: M \in SelectedNodes \Leftrightarrow N \in SelectedNodes$
- $(M, N) \in MUTEXes: (M \in SelectedNodes \Rightarrow N \notin SelectedNodes) \wedge$
$(M \notin SelectedNodes \Rightarrow N \in SelectedNodes)$

Now, we go through the types of tasks. Since the serial task introduces an ordering among its children, we can express that by the precedencies among them. However, we cannot use the set *Precedencies* because this set contains only the custom constraints, not the implicit ones.

- $T \in SerialTasks \cap SelectedNodes$:
$$\exists (C_1, \dots, C_n)\ such\ that\ \{C_i | i \in [1, n]\} = Children(T) \wedge$$
$$\forall j \in [2, n]: End(C_{j-1}) \leq Start(C_j)$$

There is no such restriction among the children of parallel or alternative tasks. Nevertheless, there are still implicit constraints related to an occurrence of tasks in the resulting schedule.

- $T \in SerialTasks \cup ParallelTasks$:
$$T \in SelectedNodes \Leftrightarrow SelectedChildren(T) = Children(T)$$
- $T \in AlternativeTasks$:
$$T \in SelectedNodes \Leftrightarrow |SelectedChildren(T)| = 1$$
- $T \in Tasks$:
$$T \notin SelectedNodes \Leftrightarrow SelectedChildren(T) = \emptyset$$

We should not forget about pins. *Pinned* activity A is a scheduled activity which cannot be moved during the run of the repair algorithm suggested in this work later on. It means that the A's temporal data is fixed.

- $Pinned = \{A \in SelectedActivities | A\ is\ pinned\}$
- $Start(A)_{before} = Start(A)_{after}$

where *before* (*after*) indicates a state before (after) the run of the repair algorithm.

Finally, we formally introduce terms selected and alternative reservation. Reservation *R* is *selected* if it performs an associated scheduled activity *A* in the resulting schedule, i.e. *Performs(A, R)*. *Alternative* reservations are those which are not selected.

- $SelectedReservations(A)$
$$= \begin{cases} \{R \in Reservations(A) | Performs(A,R)\} & if\ A \in SelectedActivities \\ \emptyset & otherwise \end{cases}$$
- $AlternativeReservations(A) = Reservations(A) \backslash SelectedReservations(A)$

Exactly one reservation from each resource group of the scheduled activity *A* can be selected.

- $A \in SelectedActivities$:
$$\forall G \in ResourceGroups(A): |G \cap SelectedReservations(A)| = 1$$

All available resources in the schedule *S* are *unary*, i.e. they can perform at most one activity at a time.

- $R \in SelectedReservations(A) \cap SelectedReservations(B)$:
$$End(A) \leq Start(B) \vee$$
$$End(B) \leq Start(A)$$

3.2 All Constraints in the Model

In the previous chapter we familiarized with all objects in the model of the Gantt Viewer's schedule and which purpose they had there. We might notice that there were described some types of constraints explicitly (e.g. precedencies), but also implicit constraints related to certain schedule's object (e.g. how the temporal data of tasks are determined). Now, we will summarize all constraints in the model and define when the schedule is feasible and when it is infeasible.

Some of the following constraints restrict a mutual position of two nodes. Since we would like to have all of them expressed with using a common construct, we introduce a *temporal constraint* which looks like:

$$a \leq y - x \leq b$$

where *x* and *y* are important points of time of the schedule (e.g. the start time of a node) and *a* and *b* are constants. We will call the former *time points* and will use a more comfortable notation *Dist(x, y)* of the constraint defined as:

$$\boldsymbol{Dist(x,y) = [a,b] \Leftrightarrow (a \leq y - x \leq b)}$$

In the schedule there can be three types of time points:

- *Start(N)* for $\forall N \in Nodes$

- *End(N)* for $\forall N \in Nodes$
- *ZeroTime* which denotes a point of time when the schedule begins; hence *ZeroTime* = 0.

In the following list of all constraints we will not distinguish between point of times and time points, i.e. $Min_{C \in Children(T)}\big(Start(C)\big)$ can be either the minimal start time among all children of the task T, or the time point which holds this minimal time. The proper type will be always clear from a particular context.

1. $\boldsymbol{T \in SerialTasks \cup ParallelTasks}$:
$$T \in SelectedNodes \Leftrightarrow SelectedChildren(T) = Children(T)$$

2. $\boldsymbol{T \in AlternativeTasks}$:
$$T \in SelectedNodes \Leftrightarrow |SelectedChildren(T)| = 1$$

3. $\boldsymbol{N \in SelectedNodes} \Leftrightarrow Parent(N) \in SelectedNodes$

4. $\boldsymbol{A \in SelectedActivities}$:
$$Dist\big(Start(A), End(A)\big) = [Duration(A), Duration(A)]$$

5. $\boldsymbol{T \in SerialTasks} \cap SelectedNodes$:
$$\exists(C_1, \dots, C_n)\ such\ that\ \{C_i | i \in [1,n]\} = Children(T) \wedge$$
$$\forall j \in [2,n]: Dist\left(End\big(C_{j-1}\big), Start\big(C_j\big)\right) = [0,\infty] \wedge$$
$$Dist\big(Start(T), Start(C_1)\big) = [0,0] \wedge$$
$$Dist\big(End(T), End(C_n)\big) = [0,0]$$

6. $\boldsymbol{T \in ParallelTasks} \cap SelectedNodes$:
$$\forall C \in Children(T): Dist\big(Start(T), Start(C)\big) = [0,\infty] \wedge$$
$$\forall C \in Children(T): Dist\big(End(C), End(T)\big) = [0,\infty] \wedge$$
$$Dist\left(Start(T), Min_{C \in Children(T)}\big(Start(C)\big)\right) = [0,0] \wedge$$
$$Dist\left(End(T), Max_{C \in Children(T)}\big(End(C)\big)\right) = [0,0]$$

7. $\boldsymbol{T \in AlternativeTasks} \cap SelectedNodes, C \in SelectedChildren(T)$:
$$Dist\big(Start(T), Start(C)\big) = [0,0] \wedge$$
$$Dist\big(End(T), End(C)\big) = [0,0]$$

8. $(\boldsymbol{M,N}) \in \boldsymbol{Precedencies}$:
$$M \in SelectedNodes \wedge N \in SelectedNodes \Rightarrow Dist\big(End(M), Start(N)\big) = [0,\infty]$$

9. $(\boldsymbol{M,N}) \in \boldsymbol{SS}$:
$$M \in SelectedNodes \wedge N \in SelectedNodes \Rightarrow$$
$$Dist\big(Start(M), Start(N)\big) = [0,0]$$

10. $(\boldsymbol{M,N}) \in \boldsymbol{SE}$:
$$M \in SelectedNodes \wedge N \in SelectedNodes \Rightarrow Dist\big(Start(M), End(N)\big) = [0,0]$$

11. $(\boldsymbol{M,N}) \in \boldsymbol{ES}$:

$$M \in SelectedNodes \wedge N \in SelectedNodes \Rightarrow Dist\big(End(M), Start(N)\big) = [0,0]$$

12. $(\boldsymbol{M}, \boldsymbol{N}) \in \boldsymbol{EE}$:

$$M \in SelectedNodes \wedge N \in SelectedNodes \Rightarrow Dist\big(End(M), End(N)\big) = [0,0]$$

13. $(\boldsymbol{M}, \boldsymbol{N}) \in \boldsymbol{Implications}$:

$$M \in SelectedNodes \Rightarrow N \in SelectedNodes$$

14. $(\boldsymbol{M}, \boldsymbol{N}) \in \boldsymbol{Equivalences}$:

$$M \in SelectedNodes \Leftrightarrow N \in SelectedNodes$$

15. $(\boldsymbol{M}, \boldsymbol{N}) \in \boldsymbol{MUTEXes}$:

$$(M \in SelectedNodes \Rightarrow N \notin SelectedNodes) \wedge$$
$$(M \notin SelectedNodes \Rightarrow N \in SelectedNodes)$$

16. $\boldsymbol{R} \in \boldsymbol{SelectedReservations}(\boldsymbol{A}) \cap \boldsymbol{SelectedReservations}(\boldsymbol{B})$:

$$Dist\big(End(A), Start(B)\big) = [0, \infty] \vee$$
$$Dist\big(End(B), Start(A)\big) = [0, \infty]$$

17. $\boldsymbol{A} \in \boldsymbol{Pinned}$:

$$Dist\big(ZeroTime, Start(A)_{after}\big) = Dist\big(ZeroTime, Start(A)_{before}\big)$$
$$= \big[Start(A)_{before} - ZeroTime, Start(A)_{before} - ZeroTime\big]$$

Note that the constraint 17 is different from the others. It is relevant just in case the repair algorithm suggested in this thesis is running; then the constraint introduces a requirement to potential resulting schedules. Otherwise, the constraint stands, but introduces nothing new to the schedule, i.e. the start time of *A* equals always the start time of *A*.

Feasible and Infeasible Schedule

The schedule *S* is *feasible* (= *consistent*) if it satisfies all 17 constraints listed above. *Infeasible* (= *inconsistent*) schedules are defined as a complement to feasible ones. In other words, if the schedule does not satisfy any of those 17 constraints, it is infeasible.

4. Existing Solutions in Visualization and Interactivity

This chapter discusses existing applications which can manage schedules and in which we found some inspiration for the Gantt Viewer as well as for the whole FlowOpt project.

4.1 iGantt

This thesis is a loose follow-up of the bachelor thesis [12] which introduced a Gantt Viewer application; in the rest of the thesis, we will call this application *iGantt* in order to distinguish it from the FlowOpt's module.

iGantt is suitable for creation of simple schedules from scratch, i.e. the user can add, edit and remove activities and resources as well as constraints and reservations. Moreover, it provides a drag & drop interactivity, highlighting of constraint violations, zooming and a repair algorithm. The last issue is discussed more in the chapter Existing Approaches in Repairing. When the application was finished and presented on some sessions, we got quite positive feedbacks despite its simplicity. Thus, we decided that we would make its successor.

In the iGantt application, there are no tasks and the only type of available constraints is precedencies. In the terminology of schedules presented in the chapter Formal Definition of the Model, all activities in iGantt are root nodes. Since iGantt's schedules are created from scratch and there is no association to workflows like in the FlowOpt project, there are neither alternative branches nor alternative reservations. In other words, all activities and all reservations are selected. In the following figures we can see a schedule for manufacturing a piston designed in the iGantt application. It is not the same as one created via the FlowOpt project (see Figure 21 and Figure 20), but it is as similar as possible.

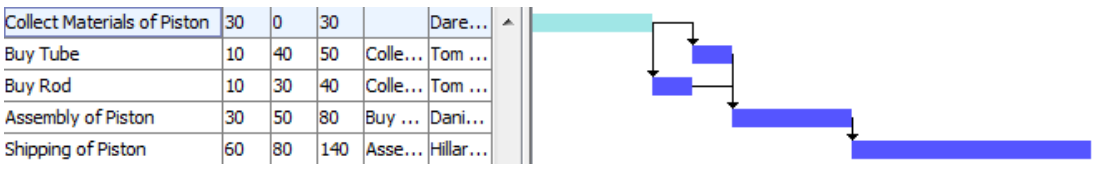


Figure 6: Schedule of manufacturing of a piston in the Task view of iGantt. In comparison with Figure 21, there are no tasks, logical constraints and pins.

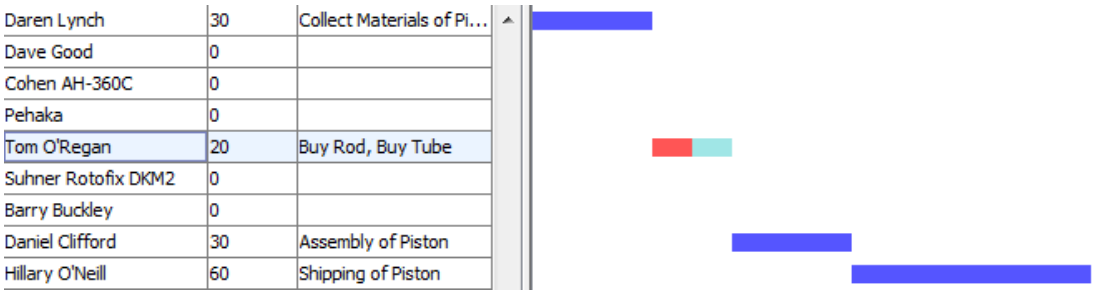


Figure 7: Schedule of manufacturing of a piston in the Resource view of iGantt. In comparison with Figure 20 there are no alternative reservations and resources which should be banned are completely removed.

4.2 MAK€ and ILOG Gantt library

The second application which had a great impact to the whole FlowOpt project was a *MAK€* application. All FlowOpt's modules including the Gantt Viewer has been integrated to *MAK€* using its plugin system and they can be used as alternatives to the MAK€'s modules since the MAK€ application has also an editor for workflows, a scheduling engine and a visualizer of Gantt charts. The only module which is quite new to MAK€ is the Schedule Analyzer.

When we take a closer look to the MAK€'s visualizer of Gantt charts, we can notice that it is not interactive. It uses a third-party library from IBM called *ILOG Gantt*, but only for displaying these charts, though the library – we should rather say a framework – provides interactivity, resource leveling (= correction of resource constraints) and many types of views of the loaded schedule. For instance in the Detail Gantt view in the library's tutorial application called Project Editor, one can see the critical path of the schedule as well as slacks of all non-critical activities.

Anyway, there are two advantages of the MAK€'s approach. The MAK€'s schedule is always compatible with associated workflows and it need not to deal with the violations introduced to the schedule by the user. Our idea was to save the first mentioned advantage in our interactive visualizer, but we wanted to allow the user to make the schedule temporarily infeasible.

Other important feature of the library is that it supports the hierarchical structure of nodes and precedence and synchronizing constraints as well as a calendar and a zooming; plus it is well designed and documented. Nevertheless, neither alternative branches and reservations, nor logical constraints are supported there.

4.3 Summary

The presented solutions are not the only ones which exist, but the presented ones influenced the Gantt Viewer the most. Microsoft Project is one of the other applications which deal with Gantt charts. It is robust, has a nice user-friendly GUI and many options, even some repair tools. The design of a window separated into two parts used in the Microsoft's application was a pattern which was utilized in the iGantt's GUI. Anyway, unlike the ILOG Gantt library, the Microsoft Project is a

standalone application which is meant for end users and cannot be used as a basement for creation of a new application.

Therefore, we decided to build our Gantt Viewer upon the IBM's library since it contained most of the functionality we wanted to provide the user in our visualizer. It saved us much work and we had not to implement everything from scratch, though the customization of the library and its extension was not that easy.

5. Visualization and Interactivity

In this chapter we will familiarize with the visualization of the Gantt Viewer's schedules and especially with their new features which the existing applications discussed in the previous chapter did not have. Moreover, we will introduce the key operations that we will be able to do in the module.

We should say that this chapter is not a user documentation of the Gantt Viewer. A few parts are similar, but the focus here is not on GUI, but on the solutions how the schedule is visualized and controlled.

5.1 Visualization

Let us start with the operations which do not modify the shown schedule, but enable the user to control its visualization.

5.1.1 Views

The Gantt Viewer provides two views of the schedule, *a Gantt Chart* and *a Resource View*. They are independent of each other in terms of existence; the Gantt Chart can exist without the Resource View and vice versa. However, both of them are working with the same schedule and their charts are synchronized. In other words, if we are currently in the Gantt Chart (resp. the Resource View) and we load a stored schedule, we will see it also in the Resource View (resp. the Gantt Chart) if we open this view. We can switch between them arbitrarily.

Both views consist of two parts, the left one and the right one. The left part is formed by the hierarchy of tasks and activities in the Gantt Chart and by the table with data about resources in the Resource View. In the right parts of both views, there are appropriate charts with calendar bars on the top.

Gantt Chart

This view is *task/activity oriented*. It means that each row represents one task, or one activity. In the left part, the tasks can be recognized according to the symbols in front of their names. If there is ⊞ (⊟), an appropriate task is collapsed (expanded). We can click on it and expand (collapse) a particular task. Activities have no such symbols. In the chart (the right part of the view), tasks' shapes start and end with the ⯆ symbol; activities' are simple rectangles.

In the Gantt Chart, all types of constraints can be shown. All of them except the resource one are visualized as arcs which connect two nodes and they differ from each other in arrows at the ends of arcs and labels above the arcs. *Precedencies'* arcs have just one arrow which points to the successor of the other node and have no label. Both *synchronizations'* and *logical constraints'* arcs have two arrows, one on

each end of the arc. You can distinguish them according to the labels which also provide more details about them. For synchronizations, there exist four possible labels which belong to four possible types, i.e. *SS*, *SE*, *ES* and *EE*. For logical constraints, it is similar, i.e. there are three labels: ⇔ (equivalence), ⇒ (implication) and *MUTEX*.

This view also visualizes *alternative branches* (the dimmed ones) and provides us the possibility to choose one of them and make the chosen branch the *selected one* (see Swapping of Reservations). We can also disable some alternative branches here (see Banning) or pin some activities (see Pinning). Branch which has been disabled cannot be selected.

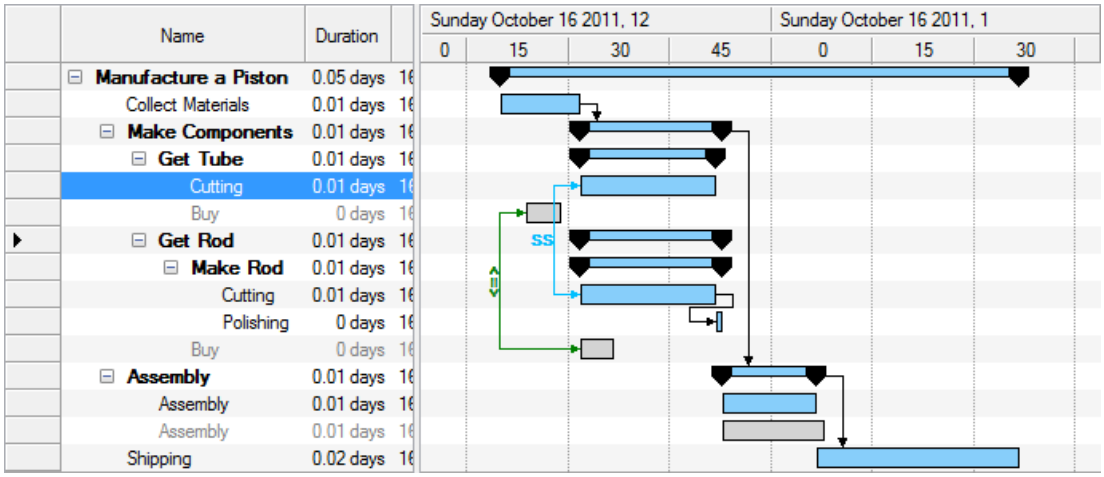


Figure 8: Gantt Chart

Resource View

The second view called a Resource View is *resource oriented*. It means that each row represents one resource.

There are no tasks and no constraints shown in the Resource View. Note that it is not even possible to correctly visualize them there since there are no activities displayed, but their reservations. We can notice it on Figure 9 which visualizes an activity "Cutting Tree" in the Resource View.

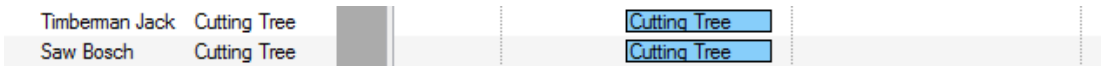


Figure 9: Activity "Cutting Tree" needs both a logger Jack and a saw Bosch (which he will use) for its performance.

Assume that the activity "Cutting tree" would be connected with some other activities via constraints. In such a case, it would not be clear which selected reservation of the activity "Cutting tree" should be connected to links of those constraints. For visualization of tasks, the argument is pretty much the same. We should add that this functionality is supported neither in the IBM's library, the Microsoft Project, nor in iGantt.

As it has already been said, there are reservations shown in the Resource View. Their shapes are same as activities – rectangles. In addition to that, inside these rectangles there is written a name of an associated activity. If there are some alternative resources to one which is allocated (= selected), we can make the allocated one alternative and an alternative allocated (see Swapping of Reservations).

Finally, in order to preserve the compatibility of the schedule with associated workflows, it is not possible to add, edit or remove a resource. However, if we wish to make a particular resource unavailable at all, we can ban it (see Banning). This operation can be rollbacked.

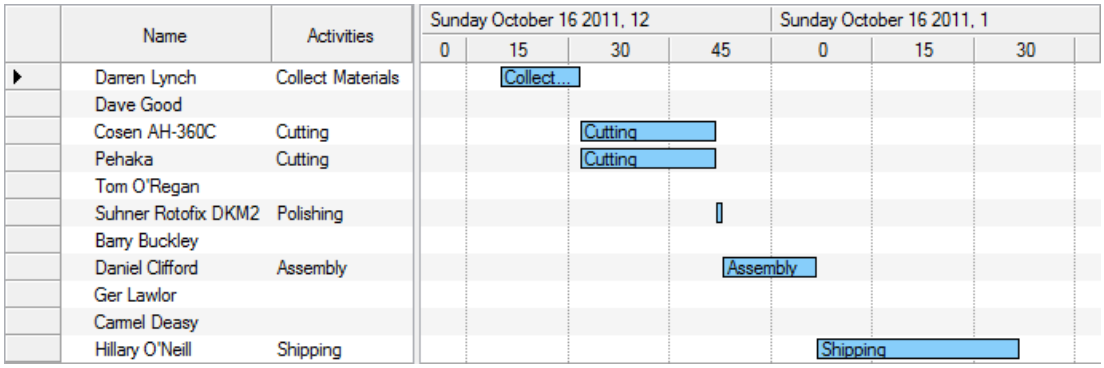


Figure 10: Resource View

5.1.2 Highlighting of Order

Before we describe how orders are visualized, we should say that the term *order* is a synonym to "an instance of a workflow". Since the Gantt Viewer has information from the Workflow Editor which order a particular node belongs to, it is possible to highlight all scheduled nodes which belong to the same *order* as the focused node. The focused node can be even an alternative one.

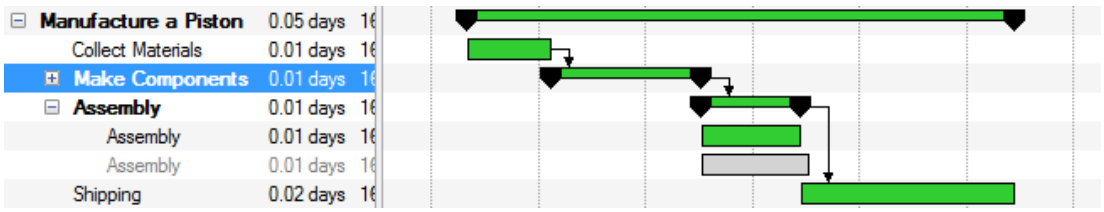


Figure 11: Focused task "Make Components" belongs to an order "Manufacture a Piston". Thus, the whole order is highlighted with the green color.

5.1.3 Highlighting of Violated Constraints

In both views, the user has an option to highlight the violated constraints like precedencies, synchronizations, logical constraints and resource constraints. If one of the first three is violated, the whole constraint's link is colored (with the red color). Moreover, parts of the nodes of the violated precedencies and synchronizations which cause these conflicts are highlighted as well (see the following figures).

Figure 12: Activities are connected with a precedence constraint, hence the lower activity should not start earlier than the upper ends, i.e. they should not overlap. However, they overlap and the precedence is violated. The overlapping parts of both activities are highlighted with the red color as well as the link of the constraint.

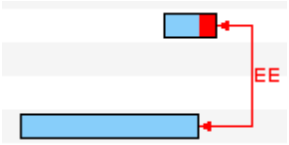


Figure 13: Violated EE synchronization. Note that exactly one node of those two connected with a violated synchronization is highlighted with the red color.

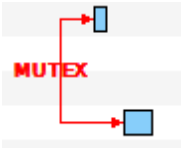


Figure 14: Violated MUTEX logical constraint. In order to satisfy the constraint, at least one of the connected activities must not be selected.

The visualization of the violated resource constraints is similar. Parts of the reservations in the Resource View (resp. activities in the Gantt Chart) which cause the violation are highlighted (with the orange color).

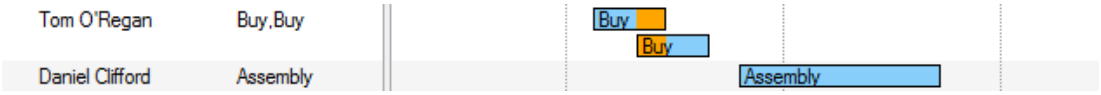


Figure 15: There are two activities "Buy" which allocate one resource. Since we consider only unary resources in this thesis, activities cannot overlap on the resources. However, the "Buy" activities do so and the resource constraint is violated, hence overlapping parts of both reservations (resp. activities) are highlighted.

Note that the row with the violated resource constraint is wider than the ordinary row

5.1.4 Filters

In the Resource View (in the Gantt Chart not), we can use two kinds of filters which we will find especially useful when the schedule is large. The first one filters out all resources except those which are associated with alternative reservations of the currently selected reservation. The second filter hides all resources except those which are allocated to an associated activity.

5.1.5 Zooming

In both described views, we can freely zoom in and out of the current schedule to gain a better perspective. There is one other zooming operation called "Zoom to fit". It zooms in/out just enough to display the whole schedule in one screen.

5.2 Tools for Schedule editing

This section describes the main functionality concerning editing of a schedule. Note that only operations which secure a backward compatibility with all associated workflows are allowed.

Nevertheless, there is one small exception. In the Workflow Editor we determine how much time each potential resource would spend on processing of a particular activity. However, as we will see in the following chapters later on, the user can change the duration of the activity and transitively influence all selected and alternative reservations, precisely their temporal data.

The second feature of the Gantt Viewer is that the user can do even such actions which violate some constraints. In other words, the consistency of the schedule is not maintained during each user's action, but it is restored on demand (see Repairing of Inconsistent Schedule).

5.2.1 Changing Temporal data of Activities

First we notice that the temporal data of alternative activities cannot be changed. It would not make much sense to provide such a functionality since we only know how long their executions are taking, but not when they are starting, or ending.

In terms of the selected activities, there are two ways how we can change their temporal data. We can either modify their temporal data directly in the table in the left part of the Gantt Chart (not in the Resource View), or we can use a mouse and make changes in the charts. In the mentioned table, there are three columns of temporal data: *Duration*, *Start Time* and *End Time*; the values in all of them can be edited. Note that the values of tasks cannot.

The second way, as we said, is to move or to resize a shape of a particular activity in the Gantt Chart, or shape of one reservation associated with the activity in the Resource View. If one reservation is changed, all the other reservations of the activity are changed as well.

Figure 16: Activity is being dragged. It is not necessary for the mouse cursor to be exactly above its row.

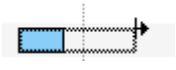


Figure 17: Activity is being resized.

The minimal supported time unit in the FlowOpt project is *one minute*.

5.2.2 Moving Tasks

The ILOG Gantt library does not support moving and resizing of tasks. The Gantt Viewer hence extends the library and adds an option to move selected tasks in order

to make the work with the application faster and more comfortable. Move of not-selected tasks would not make sense. The reason is the same as in the previous chapter. Resizing of tasks is not supported at all in the Gantt Viewer. It is not even apparent what such an action should do. However, the richest presented application in terms of this functionality – the Microsoft Project – supports both moving and resizing. The way how the Microsoft's application implements the resizing is based on a fact that temporal data of tasks are not only computed there, but also partially assigned.

In the Gantt Viewer, we can move a task via the mouse the same way as we move an activity (see Changing Temporal data of Activities). We drag its shape, move it where we want and then drop it.

Start and end time of tasks is determined by its very outside descendant activities (see Definition of Model). Thus, if we move a task, it means that all its descendant activities are moved appropriately, i.e. the same direction and by the same distance.

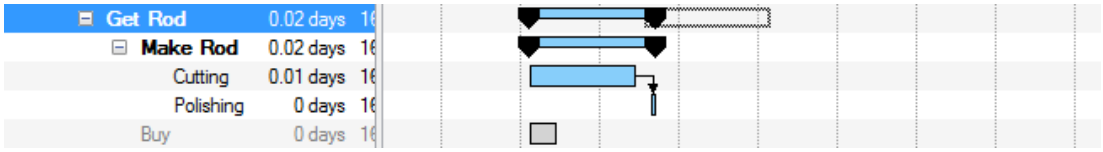


Figure 18: Task "Get Rod" with three scheduled descendants is being moved to the right. There is also an alternative activity to a task "Make Rod".

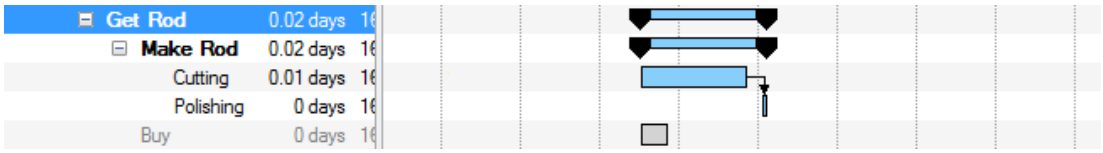


Figure 19: Task "Get Rod" has been moved. Note that an activity "Buy" has been shifted too since it is a descendant of the dragged task as well, though alternative.

5.2.3 Banning

This operation allows the user to ban either a not-selected branch, or to ban a resource which is allocated to no activity. Consequently, the user will not be able to schedule any of the banned branches (resp. to allocate banned resources). It can be useful for instance when an employee has left a company and it is necessary to introduce this fact in the schedule since there is no other possibility how to remove the resource from the schedule. The banning would make better sense if the Repair Algorithm (presented later on) took an alternative reservations, or alternative branches into account.

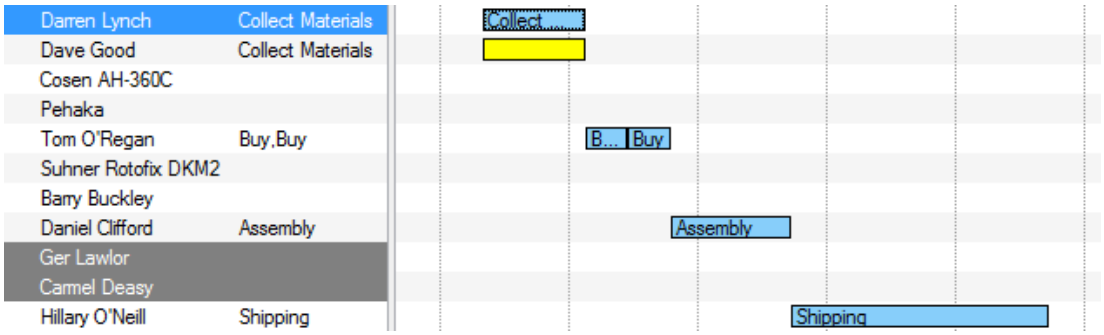


Figure 20: "Collect Materials"-"Darren Lynch" is a selected reservation here and "Collect Materials"-"Dave Good" is an alternative reservation to the selected one. Furthermore, there are two banned resources – "Ger Lawlor" and "Dave Mullins".

5.2.4 Pinning

Pinning is likely more important extension of schedules in the Gantt Viewer than banning. The pinning enables the user to place pins upon activities. As a result, pinned activities cannot be moved during the run of the repair algorithm suggested in this thesis. All the other activities can be shifted anywhere. Pinning can be applied only upon the scheduled activities; other activities and all tasks cannot be pinned.

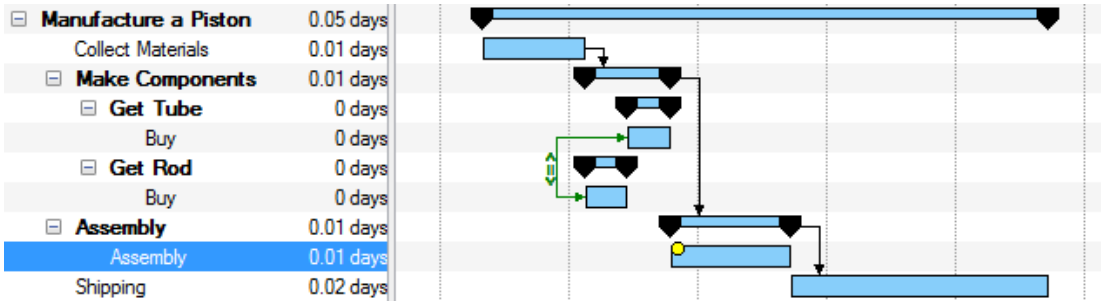


Figure 21: Activity "Assembly" is pinned and none of the others are.

5.2.5 Swapping of Branches

We can change decisions made by the Scheduler and schedule any alternative branch. In fact, we carry out swapping of a selected branch for a desired alternative one. Note that it is not possible to make a banned alternative branch the selected one. It is necessary to remove the ban first.

Figure 22: "Make" task is being made the selected branch.

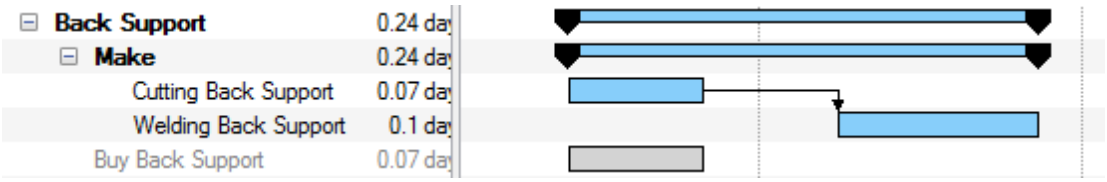


Figure 23: "Make" task has become the selected branch and "Buy Back Support" the alternative one.

5.2.6 Swapping of Reservations

Not only alternative branches can be managed by the user, but also alternative reservations. We can change the decision made by the Scheduler and swap the selected reservation for a desired alternative one in the Resource View. When we are carrying out such an operation in a large schedule we will find especially useful when we will be able to filter out the not-interesting resources (see Filters). It is evident that alternative reservations which are associated with the banned resource cannot be selected. Thus, during the process of swapping, they are not visualized.

During the swapping we can notice that all selected and alternative reservations have the same duration. This limitation is inherited from the IBM's library. In practice, it would sometimes restrict the usage of the module (e.g. one man cut the tree one minute faster than the other). Therefore, these cases are resolved earlier, i.e. during the Going from Workflow to Schedule.

Figure 24: The left mouse button has been pressed above a selected reservation "Saw 2"-"Sawing Seat".

Figure 25: The selected reservation "Saw 2"-"Sawing Seat" is being swapped with an alternative one "Saw 1"-"Sawing Seat".

Figure 26: The reservation "Saw 1"-"Sawing Seat" has become the selected one and the reservation "Saw 2"-"Sawing Seat" is now alternative. At the same time, the activity has been moved to the right.

5.2.7 Repairing of Inconsistent Schedule

The last demonstrated function of the Gantt Viewer is a repairer of violations in the schedule. The function exploits the Repair Algorithm which is presented in the following chapters; hence we omit its description now and familiarize just with the usage of the function in practice.

When we start the algorithm, it can send us a few different types of messages which say how the repairing process is going on:

1. *The optimal solution has been found.* It follows that there is no point in continuing with repairing and the algorithm ends.

2. *Solution has been found*, but not necessarily the best one. We can wait until the algorithm finds some better solution. If the algorithm meets the time limit and ends, we can try to run it once again and with the larger time limit since the algorithm starts always from the beginning. Without any doubt, the important criterion which helps us to decide whether to run the algorithm again, or not, is the rating of the best solution found till that time (see Evaluation/Rating function).

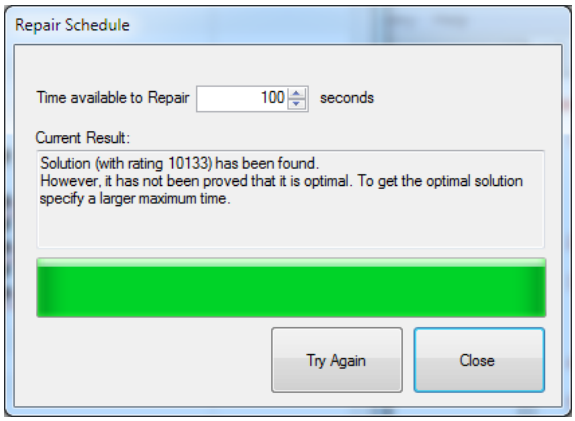


Figure 27: Solution has been found, but nobody knows whether it is the best one that exists. So, we can try to run the algorithm once again.

3. *Repair process has not started yet.* This message says that the algorithm has not even started the phase Solving of DTP. We can solve this situation by running the algorithm with the larger time limit as it is proposed for the message 2.
4. *No solution has been found till now*. The solution is the same as for the message 3.
5. *No solution exists.* This case is slightly complicated. Note that when we use the FlowOpt's Scheduler, it allows us to store a schedule just when it has found a feasible one. In other words, at the beginning we have always a consistent schedule in the Gantt Viewer. If we do some swapping or pinning carelessly, shuffling the nodes done by the repair algorithm need not to be sufficient to obtain a feasible schedule (see the following figures).

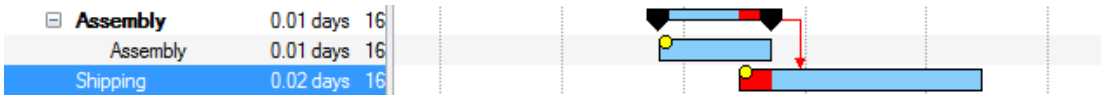


Figure 28: Schedule contains two pins. It is evident that both of them cannot be satisfied at the same time.

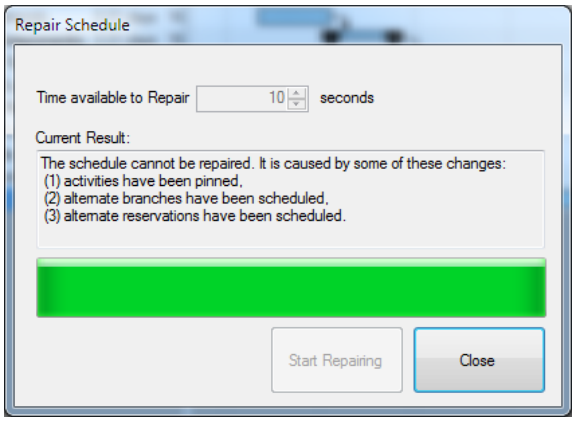


Figure 29: The repair process of a schedule in the Figure 28 failed as we expected.

6. Repair Problem

Assume there is a situation like this:

Machine breaks down and it will take a day to repair it. Thus, a production manager has to update the manufacturing schedule in a way that he postpones all activities which use that machine at that moment. In other words, the activities need to be moved to the right. The manager does that even though he may make the schedule inconsistent (the Gantt Viewer allows violations). Assume that some constraints have really been violated. The manager does not care about an inconsistent schedule, he needs a feasible one. Thus, he starts to repair these violations manually. If he is lucky, he will do some corrections and everything will be ok. However, it may also happen that one correction causes some other conflicts. In that case, the manager may spend a fair amount of time repairing the schedule and eventually get a feasible one which is completely different from the original schedule even though a more similar consistent schedule exists there.

The outlined situation should motivate us to think about an automatic solution for repairing inconsistent schedules.

6.1 Problem Description

Before we describe the problem which we are going to solve, we should introduce two terms – a *feasible* and an *infeasible* schedule – first.

6.1.1 Feasible and Infeasible Schedule

The definition of the feasible and infeasible schedules in the chapter Formal Definition of the Model is mutual for all FlowOpt's modules including the Gantt Viewer. The definition uses 17 constraints introduced in the chapter, but some of these constraints can be ignored in the viewer. The reason is that each schedule which comes out from the Scheduler and goes to the viewer is feasible (see Scheduling) and in the viewer, the user can violate only limited subset of these 17 constraints by modifications of this schedule. The ignored constraints are hence implicit constraints 1-7 except 5. Although the constraint 5 concerning the serial tasks is implicit, it introduces ordering, i.e. a set of precedencies, among the task's children and the user can violate these precedencies in the viewer. Finally, the constraint 17 is not a constraint that can be violated; actually it is a restriction on the potential solutions of the suggested algorithm. Thus, the conditions which need to be satisfied by the schedule in the Gantt Viewer are:

1. All temporal constraints (precedencies and synchronizations) among the selected nodes are satisfied (the constraints 8-12 in All Constraints in the Model, plus the implicit precedencies in the constraint 5).

2. All logical constraints are satisfied (the constraints 13-15).
3. All resource constraints are satisfied (the constraint 16).

If any of the mentioned conditions is violated, the particular schedule is infeasible.

6.1.2 Problem Definition

The problem we are going to solve in the following chapters is to find an algorithm which gets an infeasible schedule and returns a feasible one which is as similar to the original schedule as possible. There are two key questions:

- Which operations can such an algorithm carry out?
- How is the similarity defined?

Both of these questions are treated in the rest of this chapter.

6.1.3 Levels of Repair

This section will answer the first question which sounds: Which operations can the desired algorithm carry out? There are three useful operations that we can consider:

- *moving the scheduled activities anywhere in time* (abbreviated to *moving the activities*),
- *swapping the reservations*,
- *swapping the branches*.

The other operations are either redundant or not desired: moving the tasks (redundant), changing durations of the activities (not desired), removing of the temporal constraints (not desired) etc.

If we go deeply into the useful operations listed above, we get:

- *modification of the start time of the scheduled activity*.
- *making the selected reservation an alternative one plus making some other alternative reservation from the same resource group the selected one*. So, the operation concerns the change of the allocated resource.
- *making the selected branch an alternative one plus making some other alternative branch of the same task the selected one*. So, the operation concerns the change of the selected alternative in the alternative task.

Note that each of the mentioned operations introduces some particular looseness to the modification of the schedules and spaces of these three loosenesses do not intersect. In other words, we cannot get a schedule in which we have moved a single scheduled activity by any sequence of operations swapping the reservations and branches.

Then, we introduce three levels of repairing:

1. Moving the activities.
2. Moving the activities, swapping the reservations.
3. Moving the activities, swapping the reservations, swapping the branches.

From the previous discussion follows that the algorithm (*A3*) which implements the level 3 finds at least as good solution as algorithms of the level 1 (*A1*) and the level 2 (*A2*). However, the overall processing time of the *A3* is likely not shorter than the overall processing times of the algorithms *A1* and *A2* since the level 3 has the highest degree of freedom. Note that we are not saying that *A3* cannot find the optimal solution faster than the others.

On the other hand, the overall execution of the algorithm *A1* should not take more time than overall executions of the *A2* and *A3*. The trade-off for that is the quality of the resulting schedule returned by the *A1* and also the possibility that the *A1* does not find a solution at all, though the algorithms *A2* and *A3* manage that.

Allowed Operations

The algorithm suggested in this work implements only the level 1, hence the only allowed operation which the algorithm can carry out is *moving the scheduled activities anywhere in time*.

Since the level 1 does not allow swapping the branches, there is no chance how the suggested algorithm can resolve violated logical constraints. Therefore, if there is a violated logical constraint in the input infeasible schedule, the algorithm reports it and does not start until all logical constraints are satisfied. The user can correct logical constraints with the manual modifications available in the Gantt Viewer (see Swapping of Branches).

6.1.4 Evaluation/Rating function

This section discusses the second question which emerged in the Problem Definition and sounds: How is the similarity between two schedules defined? Assume that we have two schedules *S* and *T* and we can get one from the other by many applications of the Allowed Operations. Then, we can determine their similarity according to these functions:

$$f_1 = \sum_{A \in SelectedActivities} |Start_S(A) - Start_T(A)|$$

f_1: The smaller the sum of differences between the starts of the same activities is, the more similar the schedules are. It does not matter whether the activity is moved to the left, or to the right since the absolute value of the shift is added to the sum.

$$f_2 = \sum_{A \in SelectedActivities \cap \left(Start_{new}(A) \neq Start_{original}(A)\right)} 1$$

f_2: The smaller the number of activities with modified start times is, the more similar the schedules are.

There are also variations of the mentioned two functions. For instance the differences between starts in the function f_1 are raised to the power of two in order to express that two small differences are better than one large.

The algorithm suggested in this thesis uses only the function f_1 to determine how good the currently found solution is in comparison with those that the algorithm has found before.

FlowOpt's Scheduler

The FlowOpt's Scheduler does a scheduling (i.e. finding a feasible schedule from scratch), not rescheduling (i.e. repair of an existing infeasible schedule), though the Scheduler and the suggested algorithm have one in common – evaluation functions. Now we familiarize with the Scheduler's one. Since the problem which the Scheduler solves is wholly different from ours, we have to first introduce a few new terms which the evaluation function of the Scheduler works with and which we have not need till now.

Each activity *A* has a cost – *Cost(A)* – which we must pay when we execute the activity (= put the activity in the resulting schedule). It follows that the cost is a number. The simplest examples of definition of the cost function are *Cost(A) = 1* and *Cost(A) = Duration(A)*.

The schedule has a due date – *DueDate*, i.e. a point of time at which the latest node of the schedule should finish. If the due date is not satisfied, there are penalties for lateness and earliness of the schedule. The lateness is defined via two numbers *LateStep* and *LateCost*. The former determines a time interval and the latter a cost of this time interval. For instance a root node ends 5 minutes after the due date, *LateStep* = 2 minutes and *LateCost* = 20 of abstract units. Then, the cost of the delay is 50. The *EarlyStep* and *EarlyCost* are defined analogously.

Now we know all necessary terms; thus, we can introduce the evaluation function of the FlowOpt's Scheduler. The function is an addition of three sums:

$$f_3 = \sum_{R \in Roots} EarlyCost * \max\left(0, \frac{DueDate - End(R)}{EarlyStep}\right) + \sum_{R \in Roots} LateCost * \max\left(0, \frac{End(R) - DueDate}{LateStep}\right) + \sum_{A \in SelectedActivities} \left(Cost(A)\right)$$

The first sum computes all costs related to those roots which end too early. Note that there is a max function which secures that $\max\left(0, \frac{DueDate - End(R)}{EarlyStep}\right)$ is always ≥ 0. The second sum is pretty much the same as the first one, but it deals with delays. The last sum counts the cost of execution all scheduled activities. The goal of the Scheduler is then to find a feasible schedule with the smallest value of f_3.

7. Existing Approaches in Repairing

In the previous chapter the problem which is to be solved was described. The aim of this section is to survey the existing approaches related to this problem and to compare them with the algorithm suggested in this thesis later on.

7.1 Iterative Flattening Search (IFS)

First of all we will discuss an IFS [8] algorithm which consists of two stages – *relaxing* and *flattening*. During the relaxation stage the selected constraints are removed from the schedule and during the flattening stage a set of new constraints is added back to it in a way that the schedule is feasible. We should add that the constraints which are relaxed are those which have been added to the schedule by the scheduling engine (e.g. constraints for elimination of the violated resource constraints).

Since the problem we want to solve in this thesis is the repair of an inconsistent schedule, the crucial decision is how many and which constraints we should relax in order to satisfy this goal. Removing of only those constraints which are currently violated need not be enough. The reason is that the remaining constraints may restrict the schedule's nodes in a way that no feasible schedule exists there. Therefore, in the worst case it would be necessary to relax all constraints in the schedule in order to get the consistent one. The question is hence what is the smallest number of constraints we have to relax, and which are they.

Nevertheless, this approach uses a key item, a *rating function*. It can be part of the flattening stage and it says how good the current solution is and whether it is the best we have found till now. The rating function is also used in the suggested algorithm (see Evaluation/Rating function).

7.2 Right Shift Rescheduling (RSR)

Since an algorithm presented in this thesis is partly based on heuristics, we should familiarize with other heuristic-based repair techniques. One of them is the RSR algorithm [1].

When the algorithm encounters a disruption during the processing of a schedule, it postpones all activities of which executions have not been started yet by the amount of disruption time. In other words, the algorithm introduces a gap in the schedule. In the Figure 30 we can see how the RSR algorithm works illustratively. If machine *M2* breaks down during a performance of activity *1* and repair of the machine takes *r* time units, the end time of the activity will be $t + r$ instead of the original t.

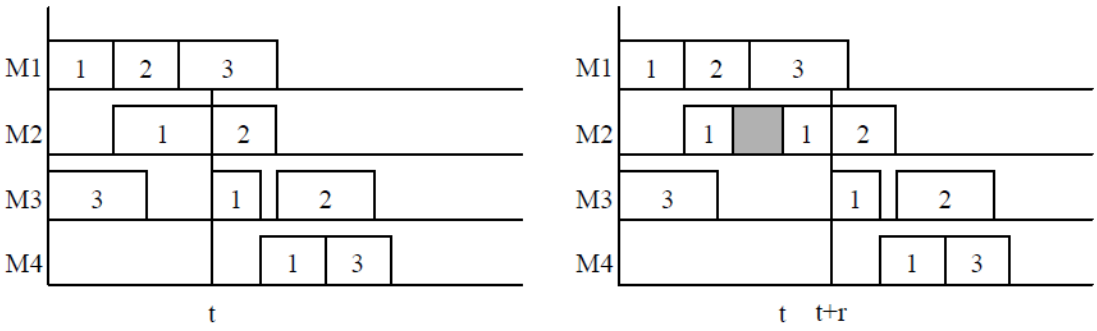


Figure 30: The RSR algorithm. Figure is from [16].

The RSR algorithm has one significant advantageous: it can immediately react to the disruption and adapt a schedule to it since the algorithm does not change anyhow already-performed activities. However, the RSR cannot be exploited for our purpose because of pins. Assume that the activity *1* on machine *M3* is pinned and the machine *M2* breaks down the same way as on Figure 30. Regardless to that, the algorithm shifts the pinned activity as well and that is not allowed. Another drawback of the RSR is that the algorithm returns likely an inefficient schedule due to the created gap.

7.3 Precedence Repair (PredRep)

PredRep algorithm [5][12] considers a different problem than the RSR but operations of both of them are similar – shifting of activities. Difference is that the RSR shifts the activities just to the right, but the PredRep shifts to both directions.

The PredRep is designed to repair schedules where the precedencies and the resource constraints have been violated manually. The algorithm works with schedules which consist of activities, unary resources, selected reservations and precedencies. There are no tasks of any type, synchronizations, logical constraints and pins in the schedules.

The PredRep has two stages – repair of the precedencies and repair of the resource constraints. The first one iterates over the violated precedencies and shifts the associated activities apart, i.e. one activity is shifted to the left and the other activity to the right. The resource constraints are not taken into account during this stage. This phase ends when all precedencies are satisfied. Let us demonstrate this part a little. Assume that $C \rightarrow D$ (as in Figure 31) is a violated precedence which is to be processed by the algorithm now. The PredRep computes how much the activities are overlapping and it shifts *C* to the left and *D* to the right according to this time. By this correction, $B \rightarrow C$ precedence is violated and hence the algorithm repairs it. More iterations may be necessary.

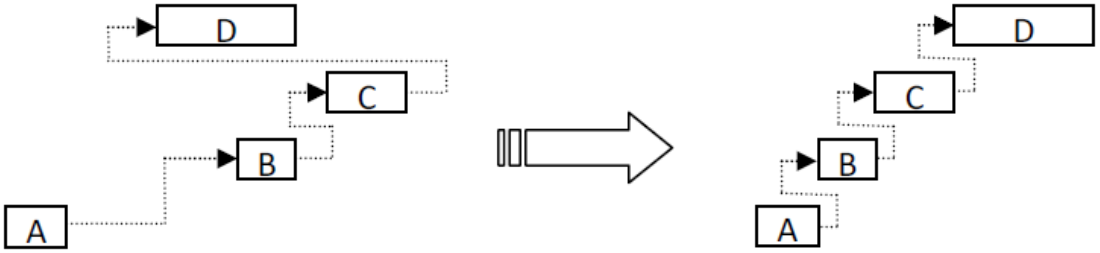


Figure 31: Input and output schedule of the PredRep algorithm. Figure is from [5].

The second stage shifts activities to the right just enough to satisfy the resource constraints of all resources. If *B* is shifted, all its successors are as well in order to not violate precedencies.

Unlike the RSR, the PredRep algorithm is not designed to repair the schedules during the schedule's execution. The reason is that it may modify parts of the schedule which precede the repairing one. However, thanks to that it returns likely better solution (i.e. more similar to the original schedule) than the RSR does.

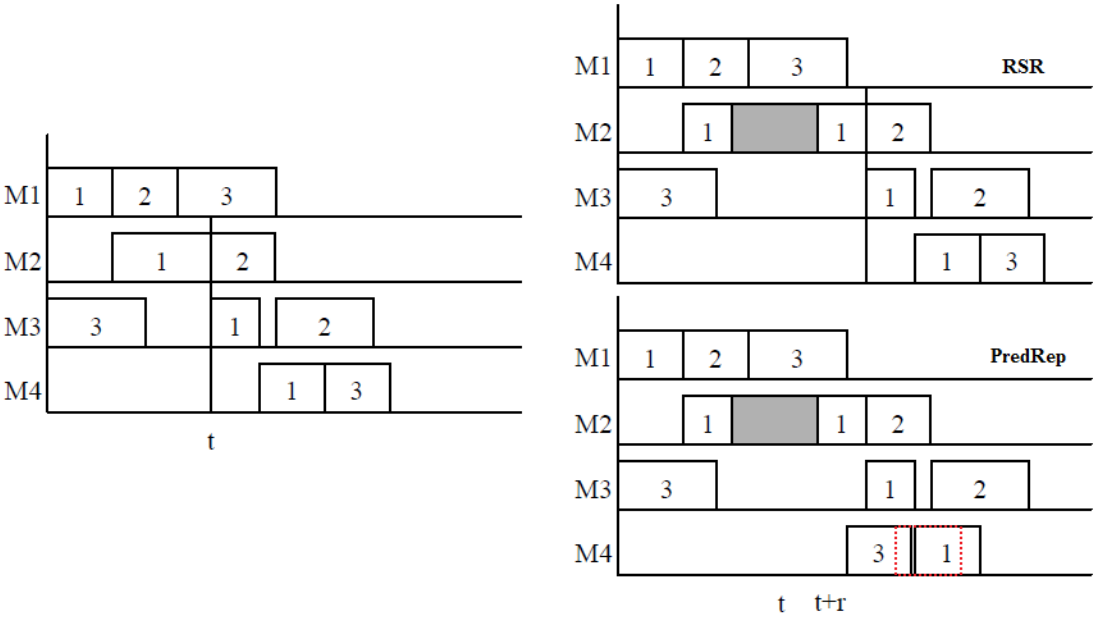


Figure 32: Comparison between the RSR and the PredRep. This figure is pretty much the same as the Figure 30, only a disruption takes longer time. In the resulting figure of the PredRep, the original position of activity 3 on machine M4 is depicted with the red dotted line and precedes the new position of activity 1 on machine M4. Therefore, the activity 3 precedes the activity 1 on the machine M4 in the new schedule and according to the Evaluation/Rating function, the schedule is more similar to the original one than the resulting schedule of RSR.

Since the repair operations in both algorithms are similar, the drawbacks of the algorithms are similar as well: neither the PredRep nor the RSR work with pins and the resulting solutions of both algorithms may not be optimal. The reason is that there is no rating function – neither in the PredRep, nor in the RSR algorithm. Nevertheless, later on we will see that the idea of the first stage of the PredRep algorithm has been found useful in the suggested algorithm.

7.4 Minimal Perturbation Problem (MPP)

Now, we sketch a more complicated problem, in particular a minimal perturbation problem [4]. The motivation for the formalization of the MPP was the situation in Purdue University, USA. There were 41 large lecture rooms and it was necessary to timetable about 1,600 meetings which took place each week. So, the coverage of those rooms was coming near 85% of the total available space. Moreover, there were constraints like the requirements and preferences of faculties and lecturers and potential student course conflicts.

The algorithm called a Limited Assignment Number (LAN) [15] was exploited for the creation such a timetable; more precisely, the algorithm tried to find a feasible solution with the maximal number of scheduled meetings. Thus, the algorithm did not guarantee that it would find the complete solution. Nevertheless, as it turned out, the LAN returned a partial solution in a reasonable time which was good enough.

The problem called a Minimal Perturbation Problem (MPP) emerged when the additional requirements came (e.g. a new class, an unexpected conference, a cancellation of the lecture). It was necessary to create a new timetable which would respect the requirements, but the impact of this change on people/meetings had to be as little as possible; hence an evaluation function f_2 introduced in the chapter Evaluation/Rating function was required; the minimal number of affected entities was looked for.

Note that the MPP problem is quite similar to ours. It is necessary to repair an existing timetable in a way that the new timetable is as similar to the original one as possible, though the similarity is defined there via a different evaluation function. Moreover, in [4] it is suggested that the LAN algorithm could be utilized as a good basement for the algorithm which would solve the MPP; the LAN would return a partial solution which the latter would modify to get a new timetable. Later on we will find out that a similar concept is used in an algorithm suggested in this thesis. First the suggested algorithm creates a new schedule *S* which satisfies all constraints – the scheduling phase – and then this feasible schedule *S* is exploited in the re-scheduling phase for obtaining another feasible schedule *T* which is more similar to the original infeasible schedule *R* than the schedule *S*.

7.5 *parc*PLAN

The last presented approach called the *parc*PLAN system is different from the others. It does not do re-scheduling, but scheduling, i.e. its result is the best schedule created from scratch, not the most similar one to the original. Nevertheless, the system is interesting for us because of the representation of the schedule's constraints which the system uses.

The goal of the system is to schedule activities on the given number of identical unary resources. In other words, an activity has no preferences in terms of resources

and it allocates any of the resources which are free. Furthermore, each pair of activities can be associated with a temporal constraint.

Before we describe how this problem is represented, we should familiarize with the graph of temporal constraints called a Simple Temporal Problem (STP) [6]. The definition of the STP below is borrowed from [9] and uses the terms introduced in the chapter All Constraints in the Model.

7.5.1 Simple Temporal Problem (STP)

An instance of the STP consists of a set of *time points* $P = \{p_1, \dots, p_n\}$ and a set of binary constraints over these time points, $C = \{c_1, \dots, c_m\}$, bounding the time distance between two time points. Every constraint $c_{i\to j}$ has a weight $w_{i\to j} \in \mathbb{Z}$ corresponding to the upper bound of the time distance between time points p_i and p_j. We can write it as an inequality $p_j - p_i \leq w_{i\to j}$. Two constraints $c_{i\to j}$ and $c_{j\to i}$ can then be combined into $-w_{j\to i} \leq p_j - p_i \leq w_{i\to j}$ giving both the upper and lower bounds. If we use the notation with *Dist*, we get $Dist(p_i, p_j) = [-w_{j\to i}, w_{i\to j}]$. If the constraint $c_{i\to j}$ ($c_{j\to i}$) does not exist, the upper (lower) bound is ∞ ($-\infty$).

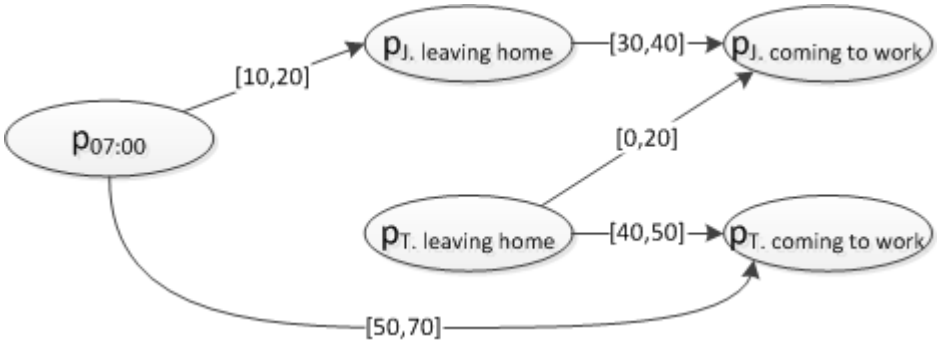


Figure 33: Example borrowed from [9]. Jerry goes to work by car which takes him 30-40 minutes and Tom goes by metro which takes him 40-50 minutes. Today, Jerry left home between 7:10 and 7:20 and Tom arrived at work between 7:50 and 8:10. The last thing which we know is that Jerry arrived at work not earlier than Tom left home, but not more than 20 minutes after that.

7.5.2 Problem's Representation in parcPLAN

The key point of the *parc*PLAN's improvement described in [7] is the representation of the problem solved there. For that purpose, the introduced STP's graph is used; vertices in the graph are activities and edges are relations among them, precisely among their temporal data. Later on, we will find out that a similar representation of relations among nodes will help us solve also our problem.

8. Repair Algorithm

This chapter describes an algorithm for solving the problem defined in the Problem Definition. There are four main sections. In the first one, we will familiarize with known algorithms and structures which are used in the suggested algorithm. Then, we will sketch the whole algorithm and in the following section, the algorithm will be described in detail and its completeness and soundness will be proved. Finally, we will propose some optimizations for the presented algorithm.

8.1 Exploited Algorithms and Structures

The suggested algorithm uses two known structures – STP and DTP – and one algorithm – IFPC. The STP has already been introduced (see Simple Temporal Problem (STP)). The DTP problem and the IFPC algorithm are described below.

8.1.1 Incremental Full Path Consistency (IFPC) algorithm

Assume we have a STP graph $G = (P, C)$ where P $= \{p_1, \dots, p_n\}$ is a set of time points and C $= \{c_1, \dots, c_m\}$ is a set of constraints (see Simple Temporal Problem (STP)). Then, we can introduce a term called *oriented path* (*transitive constraint*) together with its weight:

$$t_{i_1 \to i_h} = \left(c_{i_1 \to i_2}, \dots, c_{i_{h-1} \to i_h}\right), \forall k, l \in \{1 \dots h\}: i_k \neq i_l$$

$$w\left(t_{i_1 \to i_h}\right) = \sum_{k=1}^{h-1} w_{i_k \to i_{k+1}}$$

In other words, the path is a sequence of mutually different constraints and its weight equals the sum of weights of these constraints. Further, we say that the graph *G* has a property *all-pairs shortest path* if:

$$\forall (i, j) \text{ such that } p_i, p_j \in P \wedge i \neq j:$$
$$\exists t_{i \to j} \in G \Longrightarrow c_{i \to j} \in C \text{ such that } w_{i \to j} = Min_{s \in \{all\ t_{i \to j}\ in\ G\}} w(s)$$

So, if the graph *G* has this property, it contains not only explicitly specified constraints, but also those which transitively follow from the former.

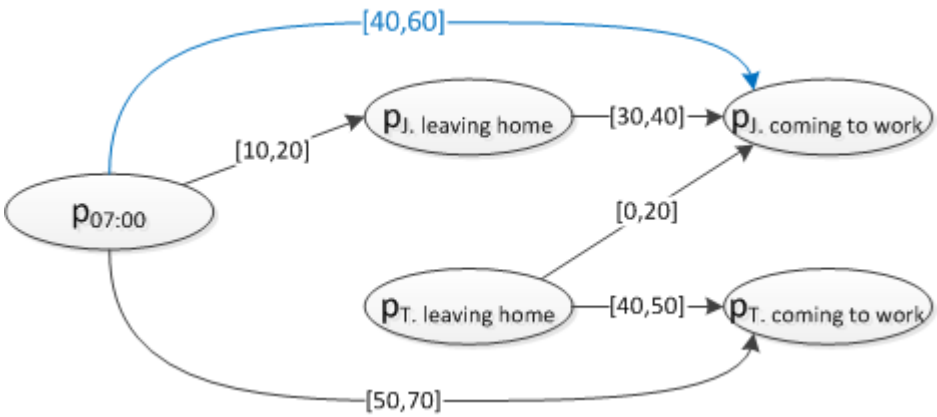


Figure 34: The STP graph from the Figure 33 enriched by a path $(p_{07:00}, p_{J.leaving\ home}, p_{J.coming\ to\ work})$. Note that the graph has not the property *all-pairs shortest path*.

IFPC [9] is an incremental graph algorithm which takes two parameters: a STP graph with the property *all-pairs shortest path* and a constraint. The IFPC algorithm adds the constraint to the graph if it is possible and secures that the modified graph fulfills the property *all-pairs shortest path* again. Since IFPC is crucial for the suggested repair algorithm, we should familiarize with the IFPC's pseudo code (borrowed from [9]):

IFPC

Input: STP graph *G = (P, C)* with the property *all-pairs shortest path*, constraint $c'_{a\to b}$.

Output: CONSISTENT if $c'_{a\to b}$ has been successfully added to G which has again the property *all-pairs shortest path*; REDUNDANT if $c'_{a\to b}$ has not been added to G since G had already contained more or same restrictive constraint; INCONSISTENT otherwise.

```
1   IF w'_{a→b} + w_{b→a} < 0 return INCONSISTENT
2   IF w'_{a→b} ≥ w_{a→b} return REDUNDANT
3   w_{a→b} ⟵ w'_{a→b}
4   I ⟵ ∅; J ⟵ ∅
5   FOR EACH p_k ∈ P, k ≠ a, k ≠ b
6       IF w_{k→b} > w_{k→a} + w_{a→b}
7           w_{k→b} ⟵ w_{k→a} + w_{a→b}
8           I ⟵ I ∪ {k}
9       END IF
10      IF w_{a→k} > w_{a→b} + w_{b→k}
11          w_{a→k} ⟵ w_{a→b} + w_{b→k}
12          J ⟵ J ∪ {k}
13      END IF
14  END FOR
15  FOR EACH i ∈ I
16      FOR EACH j ∈ J, j ≠ i
17          IF w_{i→j} > w_{i→a} + w_{a→j}
18              w_{i→j} ⟵ w_{i→a} + w_{a→j}
19          END IF
20      END FOR
21  END FOR
22  return CONSISTENT
```

Figure 35: The IFPC algorithm

In the worst case, the complexity of the algorithm is $O(n^2)$ where n is the number of points in the graph G. The complexity follows from the double loop on lines 15-21. However, the algorithm decides in the constant time whether the given constraint $c'_{a\to b}$ is consistent, redundant or inconsistent with the current graph (lines 1-2).

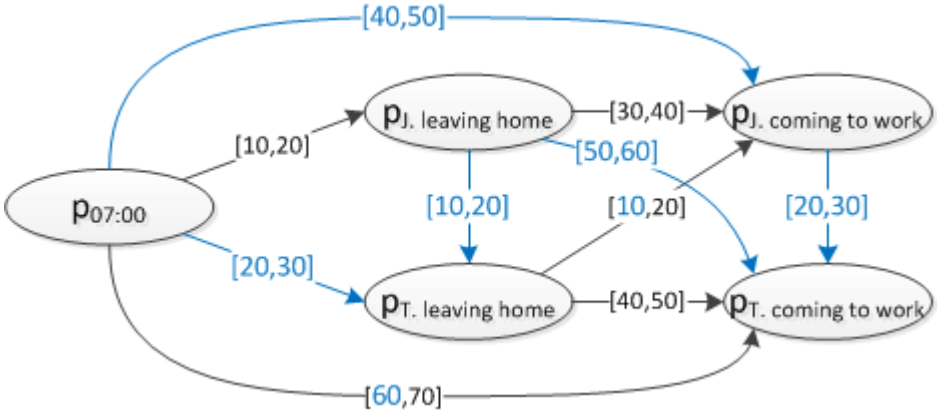


Figure 36: The STP graph from the Figure 34. Now, the graph has the property *all-pairs shortest path*. All blue temporal data have been specified or updated during the processing of the IFPC algorithm.

8.1.2 Disjunctive Temporal Problem (DTP)

The expressiveness of the introduced STP problem is not sufficient for out repair problem. We cannot formulate some constraints listed in the chapter All Constraints in the Model with the STP since the STP does not support alternatives. Those constraints are the constraint 6 which says:

$T \in ParallelTasks \cap SelectedNodes$:

$$\forall C \in Children(T): Dist\big(Start(T), Start(C)\big) = [0, \infty] \wedge$$
$$\forall C \in Children(T): Dist\big(End(C), End(T)\big) = [0, \infty] \wedge$$
$$\boldsymbol{Dist}\left(\boldsymbol{Start}(\boldsymbol{T}), \boldsymbol{Min}_{\boldsymbol{C}\in \boldsymbol{Children}(\boldsymbol{T})}\big(\boldsymbol{Start}(\boldsymbol{C})\big)\right) = [\mathbf{0}, \mathbf{0}] \wedge$$
$$\boldsymbol{Dist}\left(\boldsymbol{End}(\boldsymbol{T}), \boldsymbol{Max}_{\boldsymbol{C}\in \boldsymbol{Children}(\boldsymbol{T})}\big(\boldsymbol{End}(\boldsymbol{C})\big)\right) = [\mathbf{0}, \mathbf{0}]$$

and the constraint 16:

$R \in SelectedReservations(A) \cap SelectedReservations(B)$:

$$\boldsymbol{Dist}\big(\boldsymbol{End}(\boldsymbol{A}), \boldsymbol{Start}(\boldsymbol{B})\big) = [\mathbf{0}, \infty] \vee$$
$$\boldsymbol{Dist}\big(\boldsymbol{End}(\boldsymbol{B}), \boldsymbol{Start}(\boldsymbol{A})\big) = [\mathbf{0}, \infty]$$

We can notice that the emphasized (in bold) parts are created by disjunctions. Therefore, we use a DTP which extends the STP by allowing disjunctions. The formal definition is borrowed from [14]:

The DTP is a tuple *(P, C)* where *P* is a set of time points and $C = \{C_1, \dots, C_m\}$ is a set of disjunctive constraints (disjunctions). Each C_i is formed with a disjunction $c_{i,1} \vee \dots \vee c_{i,i_k}$ which means that C_i has i_k alternatives. Each of these alternatives is a constraint $c_{i,j,a\to b}$ known already from the STP which says that $p_b - p_a \leq w_{a\to b}$. Solution of the STP is an assignment to each time point that all constraints $c_l \in C$ are

satisfied. Solution of the DTP is defined pretty much the same way – it is an assignment to each time point that all disjunctions $C_l \in C$ are satisfied. It follows that at least one constraint in each disjunction has to be satisfied.

Now, we are able to express the constraints 6 and 16 via the DTP problem. Assume we have a parallel task *T* with two children *C* and *D*. Then, the constraint 6 will look:

$$
\begin{aligned}
&Dist\big(Start(T), Start(C)\big) = [0,\infty] \wedge \\
&Dist\big(Start(T), Start(D)\big) = [0,\infty] \wedge \\
&Dist\big(End(C), End(T)\big) = [0,\infty] \wedge \\
&Dist\big(End(D), End(T)\big) = [0,\infty] \wedge \\
&\big(Dist\big(Start(T), Start(C)\big) = [0,0] \vee Dist\big(Start(T), Start(D)\big) = [0,0]\big) \wedge \\
&\big(Dist\big(End(T), End(C)\big) = [0,0] \vee Dist\big(End(T), End(D)\big) = [0,0]\big)
\end{aligned}
$$

We can notice that there are four simple constraints and two disjunctive constraints related to the task *T*. Note that if the number of children of a parallel task equals *n* (>1), the number of simple constraints is *2n* and the number of disjunctions is still two. Those two disjunctions have always *n* constraints. This fact will be utilized in proofs later on.

Assume we have a resource *R* and there are two activities *A* and *B* which are assigned to *R*. The formulation of the constraint 16 stays the same as in the chapter 3.2:

$$
\begin{aligned}
&Dist\big(End(A), Start(B)\big) = [0,\infty] \vee \\
&Dist\big(End(B), Start(A)\big) = [0,\infty]
\end{aligned}
$$

Note that if there are *n* activities scheduled on a single resource, there are $\binom{n}{2}$ disjunctions and all of them have two constraints. The number of disjunctions can be computed as the number of all subsets of size two since the order of constraints in a disjunction is not important.

8.2 Sketch of Repair-DTP

Before we start describing the suggested algorithm, we should briefly recapitulate algorithm requirements specified in the chapter Problem Definition.

The desired algorithm gets an infeasible schedule on the input. The input schedule can contain three types of violations: violated precedencies, violated synchronizations and violated resource constraints. If the schedule contains any other type, the algorithm does not start. The only allowed operation which the algorithm can carry out is moving selected activities in time. The algorithm returns a feasible schedule as similar to the input infeasible one as possible. The similarity is defined by the function

$$f_1 = \sum_{A \in SelectedActivities} \left| Start_{output\ schedule}(A) - Start_{input\ schedule}(A) \right|$$

Though the most similar schedule is required in the assignment, we should say before we start to present the suggested algorithm called a *Repair-DTP* that the algorithm *does not guarantee* the return of the most similar schedule to the input one which exists. Among others, it is caused by a chosen approach when only a selected subset of potential solutions is inspected. This topic is more discussed in the chapter Similarity later on.

The following sketch of the Repair-DTP algorithm describes only the main idea of the algorithm. The next chapters then discuss each step of the algorithm in detail.

1. First we create two time points for each selected node *N* in the input infeasible schedule and add them to an empty STP graph. One time point represents the start of *N* and the other represents the end of *N*. It follows that each time point is associated with one node and holds appropriate temporal data of the node which we call *a current time*.
    - Example: The input schedule contains an activity A which starts at January 15, 2012, 8 am and ends the same day at 10 am. Then, the STP graph contains two time points: one associated with the beginning of A and one with the end of A. The former's current time equals January 15, 2012, 8 am and the latter's one January 15, 2012, 10 am.
2. We create an artificial time point p_0 called *a zero time point* and we add it to the STP graph. Then, we connect the point p_0 with all the other points in the graph with precedencies so that p_0 is a predecessor of all the other points. The zero point's current time is set to an absolute time (e.g. January 15, 2012, 8 am) which determines the earliest allowed time in the resulting feasible schedule and is constant.
    - Note that the zero point limits the search space of potential solutions if its current time is not set to $-\infty$.
3. We express the repair problem as a DTP problem. In more detail, we take all nodes, custom constraints, reservations and pins and we convert them to simple constraints and disjunctions according to corresponding constraints introduced in the chapter All Constraints in the Model. Then, we add all new simple constraints and disjunctions to an empty DTP. One of the feasible schedules of the solution of the new DTP problem is then the solution of our repair problem. Afterwards, we take all simple constraints of the DTP and we successively add these temporal constraints to the existing STP graph. That is a moment when the IFPC algorithm is exploited.
    - Note that the IFPC algorithm secures that the added simple constraints are propagated to the rest of the STP graph. Then, the current time of

the zero point together with up-to-date weights of precedencies going to the other points determine the minimal valid times of the other points. We call these times *the minimal times*. Since each time point *p* (except the zero point) is associated with a start (end) time of a certain node *N*, the minimal time of *p* restricts the start (end) time of *N*.

- There is also *the maximal time* for each time point *p* (except the zero point) which is a counterpart of the *p*'s minimal time and which is determined by the weights of constraints in the STP graph as well. However, the difference is that there is no such "global" successor as the zero time point is the "global" predecessor. Therefore, the maximal time need not to be set. In that case, it equals the maximal allowed time in the resulting schedule.

4. We compute the minimal and the maximal time for each time point (except the zero point) and we hold these times by the point. It means that temporal data of each point (except the zero point) consist of three items: the minimal time, the current time and the maximal time. If the current time is less than the minimal one, the current time is set to the value of the minimal one. Similarly, if the current time is greater than the maximal one, the current time is set to the value of the maximal one.

    - In other words, if no feasible schedule exists with an activity *A* starting (ending) at a time t_1, but there is a chance that for a time $t_2 > t_1$ a feasible schedule exists, the activity is moved to the right and *A* newly starts (ends) at the time t_2. For the maximal time (i.e. $t_2 < t_1$), it is analogous.

5. We go through all combinations of constraints of disjunctions of the DTP problem. If we find a combination which satisfies all disjunctions and which is consistent with the STP graph, it means that we have found *a feasible minimal schedule (MS)* for the current combination; hence we continue to the step 6. If we have explored all combinations, we quit the repair algorithm and return the best schedule found till now if such a schedule exists.

    - Minimal schedule (MS) is a schedule which consists of minimal times of all time points (except the zero point).

    - Note that the feasible minimal schedule (MS) represents a potential solution to the repair problem. However, the schedule is likely not the most similar one which we can find (see the step 6).

6. We create a copy of the STP graph and add the current combination of constraints to the new graph. Then, we apply (not "go to") the step 4 (i.e. update of the current times) once again, but it does not mean that *the current schedule (CS)* is feasible.

- Current schedule (CS) is an analogy to the minimal schedule (MS) and consists of current times of all time points (except the zero point).
- Example of infeasible current schedule (CS): There are two time points *A* and *B* in the new STP graph. *A*'s minimal time = *B*'s minimal time = *B*'s current time = (*A*'s current time – 1 hour). The graph includes a constraint $Dist(A, B) = [0, \infty]$, i.e. *A* is a predecessor of *B*. It follows that the constraint is violated since *A* is later than *B* according to the current times.

7. We take a violated constraint in the current schedule (CS) if such a constraint exists, and we continue to the step 8. If the schedule is feasible, we compute its rating with using the function f_1 and if the new schedule is better than the best one found till now, we save the new schedule. Then, we continue to the step 5.
8. Assume we have time points *A* and *B*, the violated constraint looks $B - A \leq w$ and the current distance of the points is $w'(> w)$. Then, we shift *A* to the right by $\left\lfloor \frac{w'-w}{2} \right\rfloor$ and *B* to the left by $\left\lceil \frac{w'-w}{2} \right\rceil$. This operation repairs the constraint and $w' = w$.
    - The modified current schedule (CS) is less or equally similar to the input infeasible one. However, since we are shifting time points more or the same to the left as to the right, the modified current schedule (CS) is more or equally similar to the feasible minimal schedule (MS) which we found in the last processing of the step 5. If we do not find any more similar schedule by corrections of violated constraints, we have a "safety brake" in a form of the feasible minimal schedule (MS). It means that we shift time points till the time the step 7 returns the feasible minimal schedule (MS).
    - Later on we will prove that this repair operation never assigns a value less than the *B*'s minimal time to the *B*'s current time.

We can notice that in the sketch there is no reference to pins. We will see later on that they have no crucial effect to the skeleton of the Repair-DTP algorithm; hence they have been omitted till now.

8.3 Skeleton of Repair-DTP

In the previous chapters, we familiarized with the known structures and algorithms which we would need later on and with the basic idea of the suggested repair algorithm Repair-DTP. This chapter is the first one which describes the algorithm in detail and provides us with its pseudo codes. We will go through the algorithm top-down.

Temporal data of Time Points

Before we describe the "wrapping" function, we should discuss the temporal data of each not-zero time point *P* in detail. The reason is also that we have omitted pins in the sketch of the algorithm.

- *A current time* – the absolute time of the point *P* which equals the resulting start (end) time of an associated scheduled node *N* in the resulting schedule. The original value of the current time equals the start (end) time of *N* in the input infeasible schedule. During the run of Repair-DTP, the value of the current time is modified if the value is out of the range of the minimal and maximal times (the step 4 in the sketch), or if some violated constraint connected with the node *N* is being repaired (the step 8 in the sketch).
- *A minimal time* – the minimal valid value of the current time of the point *P*. At the beginning of Repair-DTP, when the zero time point is added to an STP graph (the step 2 in the sketch), the minimal time equals the zero time. The minimal value increases (the step 4 in the sketch) if a more restrictive constraint is added to the STP (the steps 3 and 6 in the sketch). One of the more restrictive can be also a pin.
- *A maximal time* – the maximal valid value of the current time of the point *P*. At the beginning of Repair-DTP, the *maximal time* equals the maximal allowed time in the resulting feasible schedule (e.g. 31/12/9999 23:59) and if the input schedule *S* contains no pins, it is not changed. If there are some pins in the schedule *S*, maximal values of some time points may decrease.

The temporal data of the zero time point consists of only one time called a *ZeroTime*. The ZeroTime is also absolute time as all the current, minimal and maximal times and determines the earliest allowed time in the resulting feasible schedule. The ZeroTime is constant.

Repair function

The wrapping function is called *Repair*. It gets an infeasible schedule *S* with no violated logical constraint as its input and returns the most similar feasible schedule to *S* among the schedules the function has found.

REPAIR

Input: infeasible schedule *S* with no violated logical constraint.

Output: the most similar feasible schedule to *S* among the found schedules.

```
1   IF dtp ← ConvertToDTP(S) fails
2      FAIL
3
4   UpdateTimes(dtp)
5
6   SolveDTP(dtp, 1)
7   IF solution of dtp exists
8      apply the best solution on schedule S
```

Figure 37: The Repair function of the Repair-DTP algorithm

The *ConvertToDTP* function on the line 1 does the step 1-3 in the sketch. It means that the function creates a DTP problem together with a STP graph and adds to the graph all simple constraints of the DTP. It may happen that the schedule S cannot be repaired (e.g. situation in Figure 28). In such a case, the Repair function fails.

Afterwards, the *UpdateTimes* function computes the minimal and maximal times of all time points according to the constraints in the STP graph and corrects the current times which are not valid (see the step 4 in the sketch).

The *SolveDTP* function is the key part of the Repair-DTP algorithm. The function finds all possible feasible minimal schedules (MS) and for each of them the function finds the most similar current schedule (CS) (see the steps 5-8 in the sketch). The function takes two parameters – the DTP problem which consists of

- the STP graph and
- the list of all disjunctions. All disjunctions have more than one constraint.

The second parameter is a number (*disjIndex*) that determines which disjunction is to be processed the next time. Later on we will see that the second parameter is necessary since the SolveDTP function is implemented recursively.

The final step of the Repair function is an application of the best found feasible schedule to the input schedule *S*. This operation has to be done explicitly since the repair algorithm does not change the schedule *S* during its run. Actually, the algorithm needs the schedule *S* only in the *ConvertToDTP* function. Afterwards, the whole problem is represented via a DTP.

Repair function – Proofs of Completeness and Soundness

Since the Repair function is a sequence of calls of other functions and the function contains no loop directly in its body, the complexity of the Repair function equals the complexity of the most complex function which the Repair function calls. The soundness of the Repair function depends on the called functions as well. In other words, both proofs have to be done on the level of sub-functions.

8.4 Conversion to DTP

This section concerns the conversion of a schedule S to an appropriate DTP problem. The function which provides the whole conversion is called ConvertToDTP.

CONVERTTODTP

Input: schedule *S*

Output: DTP representation of *S*. It consists of STP graph and the list of unresolved disjunctions.

```
1   graph ⟵ CreateTimePoints(S)
2   disjs ⟵ ∅
3   IF ConvertConstraints(graph, disjs, S) fails
4      FAIL
5   return new DTP with graph and disjs
```

Figure 38: The ConvertToDTP function of the Repair-DTP algorithm

The *CreateTimePoints* function creates a STP graph, adds all necessary time points to the graph and connects all points (except the zero point) with the zero point with precedence constraints.

The second function called *ConvertConstraints* takes all considered constraints formally described in the chapter All Constraints in the Model and converts them to simple constraints and disjunctions appropriately. Furthermore, the function adds the simple constraints to the STP graph. The ConvertConstraints function can fail since the schedule *S* can be over-constrainted (e.g. situation in Figure 28).

The proofs of the completeness and soundness of the ConvertToDTP function will be presented later on (see ConvertToDTP function – Proofs of Completeness and Soundness).

8.4.1 Initialization of Time Points

Now we take a closer look at the CreateTimePoints function which gets a schedule and returns a new STP graph with all appropriate time points. Before we familiarize with the pseudo code of the function, we introduce a command `new TimePoint(min, current)` which creates a new not-zero time point with the *minimal time* equals `min`, *current time* equals `current` and *maximal time* equals the maximal allowed time in the resulting schedule (e.g. 31/12/9999 23:59).

CREATETIMEPOINTS
Input: schedule *S*
Output: new STP graph with initialized time points

```
1   graph ⟵ new STP graph
2
3   IF ∃A|A ∈ SelectedActivities of S ∧ Pin(A)
4      ZeroTime ⟵ 01/01/0001 00:00
5   ELSE
6      ZeroTime ⟵ Min_{N∈SelectedActivities of S}(Start(N))
7   END IF
8   add zero time point with ZeroTime to graph
9
10  FOR EACH N ∈ SelectedNodes of S
11     add new TimePoint(ZeroTime, Start(N)) to graph
12     add new TimePoint(ZeroTime, End(N)) to graph
13  END FOR
14  return graph
```

Figure 39: The CreateTimePoints function of the Repair-DTP algorithm

On the lines 3-8, we create the zero time point and add it to the new STP graph. Then, we add two time points for each selected node to the graph. The `add` functions on the lines 11 and 12 also connect all not-zero points with the zero point with precedencies.

Note that there are two potential values of the ZeroTime. If there is no pin in the schedule *S* (the line 6), we need to somehow anchor the STP graph in time; hence the ZeroTime equals the start time of the earliest selected activity. It follows that all potential feasible schedules which start before this ZeroTime cannot be found by the Repair-DTP algorithm and that is the reason why we cannot guarantee finding the most similar schedule to the input infeasible schedule.

If there is at least one pin in the schedule *S* (the line 4), the pin anchors the graph in time on its own. Nevertheless, we still need the zero point for the computation of the minimal times of other points (see the step 4 in the sketch). Therefore, we set the ZeroTime to the lowest value of our time axis (e.g. 01/01/0001 00:00). In comparison with the situation when no pin is available, no potential feasible schedule which can be found by the Repair-DTP algorithm is eliminated here. In the real implementation of the algorithm, the data type used for the representation of temporal data is `long`. The `long` can hold the time 01/01/0001 00:00 as well as the current time and 31/12/9999 23:59.

We demonstrate the CreateTimePoints function on a simple example. Assume we have a schedule like this:

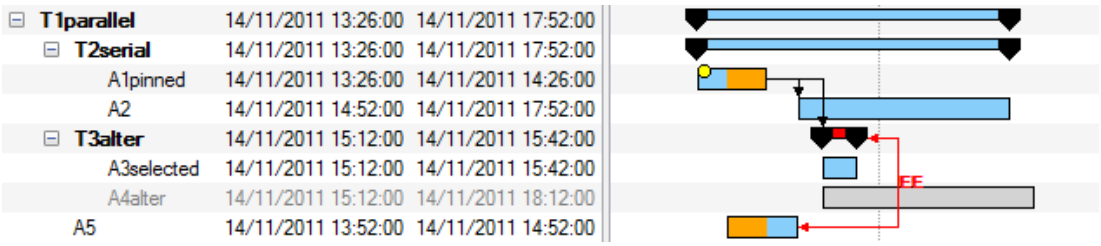


Figure 40: Infeasible schedule to repair – the Gantt Chart

Figure 41: Infeasible schedule to repair – the Resource View

Then, we get a STP graph:

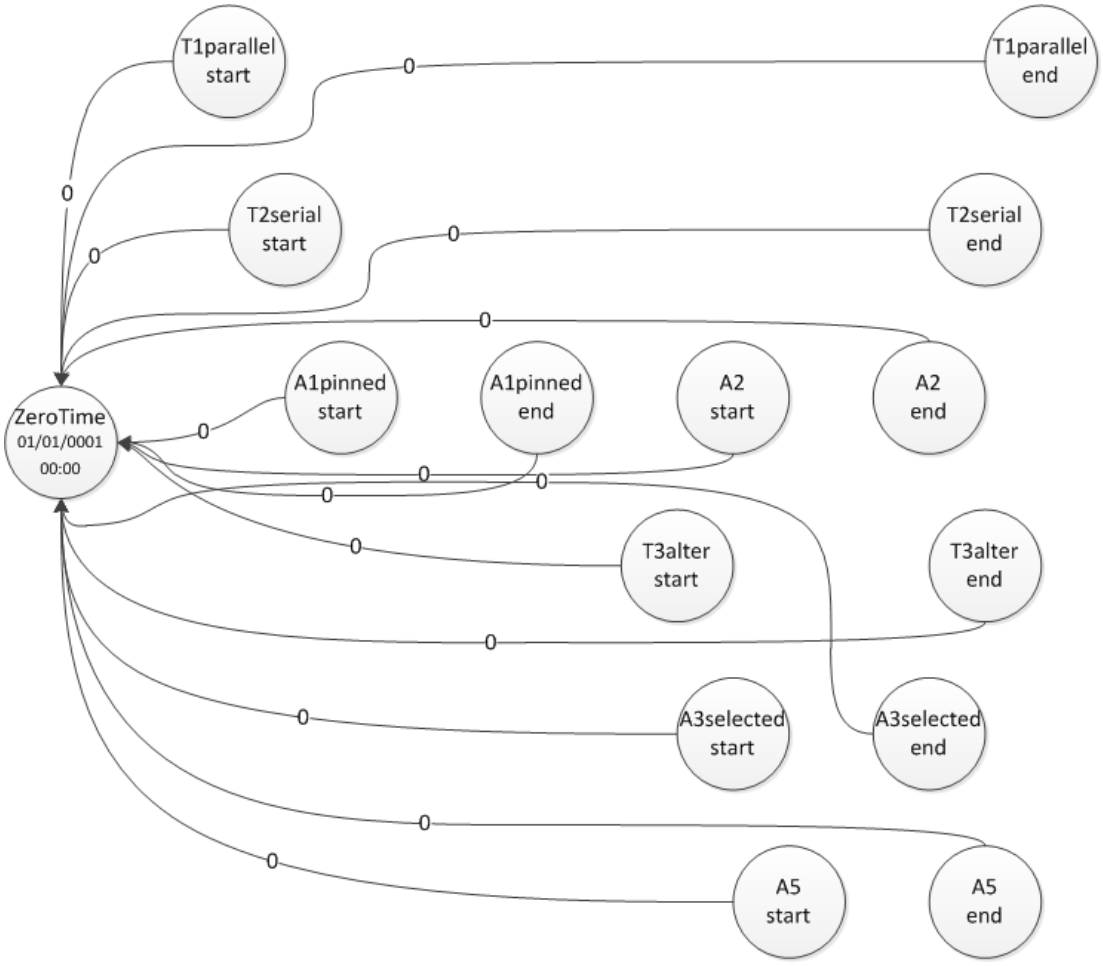


Figure 42: A STP graph returned from the CreateTimePoints function. The minimal and current times of time points are not displayed.

Note that the ZeroTime equals 01/01/0001 00:00 since the *A1pinned* activity is pinned. Moreover, there are arrows from all time points (except the zero point) to the zero point. The arrows represent the precedencies ($the\ zero\ point \rightarrow other\ point$). We can notice that arrows are pointing to the zero point since the precedencies in the

STP look like $(the\ zero\ point - other\ point \leq 0)$ (see Simple Temporal Problem (STP)).

CreateTimePoints function – Proofs of Completeness and Soundness

First we proof the completeness of the CreateTimePoints function. Operations on the lines 3 and 6 have the complexity $O(m)$ where m is the number of the selected activities in schedule *S*. The creation of time points on the lines 11 and 12 has the complexity $O(1)$, but adding of precedencies $(the\ zero\ point \rightarrow other\ point)$ to the graph has the temporal complexity $O(n^2)$ where $n\ (\geq m)$ is the number of the selected nodes. The quadratic complexity is caused by the used Incremental Full Path Consistency (IFPC) algorithm. The for-loop on the lines 10 to 13 has hence the complexity $O(n^3)$ and that is also the complexity of the CreateTimePoints function.

The soundness of the CreateTimePoints function is evident from the description the presented pseudo code and the following description. The only question which emerges is why the time points of the not-selected nodes are not added to the STP graph. The answer is that the Repair-DTP algorithm can only change starts of the selected activities, hence there is no chance how to schedule not-selected nodes (see Problem Description).

8.4.2 Conversion of Constraints

The second function used for the conversion of the input schedule *S* to an appropriate DTP problem is the ConvertConstraints function. The function has three input parameters: the schedule *S*, a STP graph *G* and an empty list of disjunctions *DI*. The function retrieves all considered constraints (see All Constraints in the Model) which are in the schedule *S* as simple constraints and disjunctions. The simple constraints are added to the graph *G* via the IFPC algorithm. The disjunctions are added to the list *DI*. The function returns the graph *G* and the list *DI*.

Note that since the Repair-DTP algorithm works only on the schedules without violated logical constraints and cannot schedule not-selected nodes, we can omit all constraints which concern these two areas (i.e. the constraints 1-3 and 13-15 in the chapter All Constraints in the Model). Such constraints have to be satisfied in the input schedule of the algorithm. We can notice that all remaining constraints are temporal.

The command `add Dist(x, y) = [a, b] to G` in the pseudo code of the ConvertConstraints function, which follows, means that the simple constraint *Dist(x, y) = [a, b]* is being added to the STP graph *G* via the IFPC algorithm. On the other hand, adding of a constraint to a disjunction is direct, i.e. no algorithm is exploited.

CONVERTCONSTRAINTS

Input: STP graph *G*, disjunctions *DI*, schedule *S*

Output: *G* and *DI* with constraints related to a hierarchy of nodes of *S*

```
1   FOR EACH N ∈ SelectedNodes of S
2      IF N ∈ Activities of S
3         t_dur = Duration(N)
4         add Dist[TimePoint(Start(N)), TimePoint(End(N))] = [t_dur, t_dur] to G
5
6         IF N ∈ Pinned of S
7            t_start = Start(N) − ZeroTime
8            add Dist[TimePoint(zero), TimePoint(Start(N))] = [t_start, t_start] to G
9         END IF
10
11     ELSE IF N ∈ SerialTasks of S
12        first = Children(N)[1]
13        add Dist[TimePoint(Start(N)), TimePoint(Start(first))] = [0,0] to G
14
15        childrenCount = |Children(N)|
16        last = Children(N)[childrenCount]
17        add Dist[TimePoint(End(N)), TimePoint(End(last))] = [0,0] to G
18
19        FOR i = 2 TO childrenCount
20           from = TimePoint(End(Children(N)[i − 1]))
21           to = TimePoint(Start(Children(N)[i]))
22           add Dist[from, to] = [0, ∞] to G
23        END FOR
24
25     ELSE IF N ∈ ParallelTasks of S
26        d_start, d_end ← ∅
27        FOR EACH C ∈ Children(N)
28           add Dist[TimePoint(Start(N)), TimePoint(Start(C))] = [0, ∞] to G
29           add Dist[TimePoint(End(C)), TimePoint(End(N))] = [0, ∞] to G
30           add Dist[TimePoint(Start(N)), TimePoint(Start(C))] = [0,0] to d_start
31           add Dist[TimePoint(End(N)), TimePoint(End(C))] = [0,0] to d_end
32        END FOR
33        add d_start and d_end to DI
34
35     ELSE IF N ∈ AlternativeTasks of S
36        C ∈ SelectedChildren(N)
37        add Dist[TimePoint(Start(N)), TimePoint(Start(C))] = [0,0] to G
38        add Dist[TimePoint(End(N)), TimePoint(End(C))] = [0,0] to G
39     END IF
40  END FOR
41
42  FOR EACH (M, N) ∈ Precedencies of S
43     IF both M,N ∈ SelectedNodes of S
44        add Dist[TimePoint(End(M)), TimePoint(Start(N))] = [0, ∞] to G
45  END FOR
46
47  FOR EACH (M, N) ∈ SS of S
48     IF both M,N ∈ SelectedNodes of S
49        add Dist[TimePoint(Start(M)), TimePoint(Start(N))] = [0,0] to G
50  END FOR
51  … // other synchronizations similarly
```

```
52
53    FOR EACH R ∈ Resources of S
54       activs ← {A|A ∈ SelectedActivities of S ∧ R ∈ SelectedReservations(A)}
55       FOR EACH (A1, A2)|(A1 ∈ activs) ∧ (A2 ∈ activs) ∧ (A1 ≠ A2)
56          d1,2 ⟵ ∅
57          add Dist[TimePoint(End(A1)), TimePoint(Start(A2))] = [0, ∞] to d1,2
58          add Dist[TimePoint(End(A2)), TimePoint(Start(A1))] = [0, ∞] to d1,2
59          add d1,2 to DI
60       END FOR
61    END FOR
62    return G and DI
```

Figure 43: The ConvertConstraints function of the Repair-DTP algorithm

As we can see, the implementation of the ConvertConstraints function is quite straightforward. For instance the constraint on the line 4 (activities) is the constraint 4 in the chapter All Constraints in the Model, the constraint on the line 8 (pins) is an application of the constraint 17 and so on. Lines 30-31 and 54-60 are the only ones which are slightly complicated; parallel tasks and reservations are converted to disjunctions there. Nevertheless, we have already discussed this conversion in the chapter Disjunctive Temporal Problem (DTP).

Now we demonstrate the function on the sample schedule depicted on the Figure 40 and Figure 41. We take the STP graph in the Figure 42 and remove all constraints (precedencies) from it in order to make the graph more transparent. During the run of the ConvertConstraints function, the constraints are added to the graph via the IFPC algorithm, but we do not do that and add the constraints to the graph directly. The reason is a good transparency again.

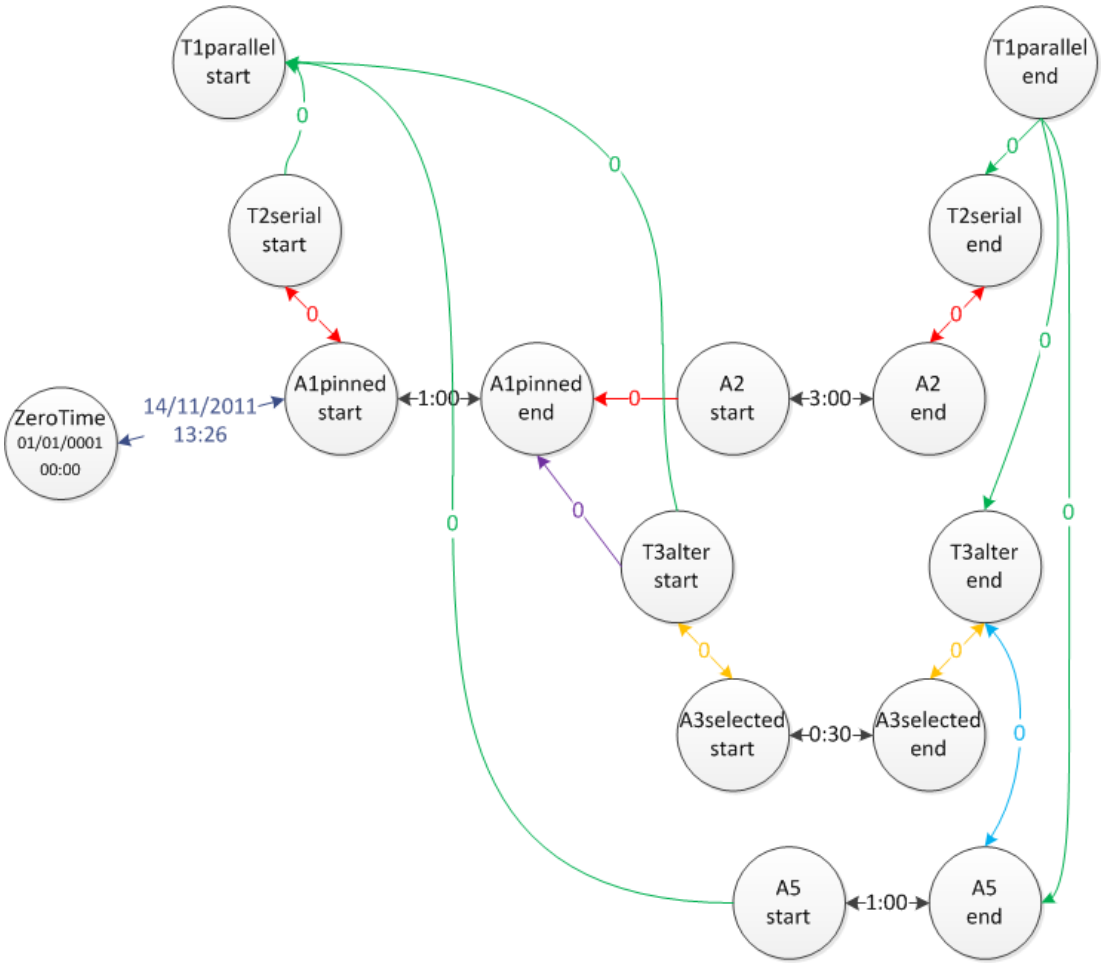


Figure 44: A STP graph returned from the ConvertConstraints function. Arrows produced by the constraint 4 (5, 6, 7, 8, 9-12, 17) have the black (red, green, orange, purple, light blue, dark blue) color. Arrows in a form $a \overset{t}{\leftrightarrow} b$ means $-t \leq b - a \leq t$, i.e. the time distance of between a and b is fixed.

We should add that the ConvertConstraints function also returns a list of disjunctions which look:

1. $Dist\big(Start(T1parallel), Start(T2serial)\big) = [0,0] \vee$
   $Dist\big(Start(T1parallel), Start(T3alter)\big) = [0,0] \vee$
   $Dist\big(Start(T1parallel), Start(A5)\big) = [0,0]$
2. $Dist\big(End(T1parallel), End(T2serial)\big) = [0,0] \vee$
   $Dist\big(End(T1parallel), End(T3alter)\big) = [0,0] \vee$
   $Dist\big(End(T1parallel), End(A5)\big) = [0,0]$
3. $Dist\big(End(A1pinned), Start(A5)\big) = [0,\infty] \vee$
   $Dist\big(End(A5), Start(A1pinned)\big) = [0,\infty]$

The first two disjunctions come from the constraint 6 (parallel tasks) in the chapter All Constraints in the Model and the last disjunction comes from the constraint 16 (reservations).

ConvertConstraints function – Proofs of Completeness and Soundness

From the pseudo code and the description of the ConvertConstraints function, it is apparent that the function is sound. The completeness of the function can be trivially proved as well since the schedule always contains the finite number of nodes, constraints and resources; all loops in the pseudo code have the finite number of iterations.

Note that the simple constraints could be also added to the STP graph *G* in a different way. First all of them would be inserted there directly, i.e. no algorithm for propagation of constraints would be used. Afterwards, in the second phase, some non-incremental algorithm would run in order to provide us with *G* with the property *all-pairs shortest path*. It would have one advantage: speed. However, we will see later on in the chapter Experiments that the conversion of the repair problem to a DTP is not a bottleneck of the Repair-DTP algorithm.

8.4.3 ConvertToDTP function – Proofs of Completeness and Soundness

It is evident that the ConvertToDTP function is sound since every constraint which is in the input schedule has been added to the new returned DTP. The completeness of the function follows from the completeness of the CreateTimePoints and ConvertConstraints functions and the fact that the pseudo code of the ConvertToDTP function contains no loop.

8.5 Update of Times

The input infeasible schedule has been successfully converted to an appropriate DTP problem in the previous chapters which corresponds to the steps 1-3 in the sketch of the Repair-DTP algorithm. This chapter focuses on the step 4 where the minimal and the maximal times of each time point (except the zero point) are computed and the current time is updated according to these bounds.

As it was already said before, the ZeroTime and all current, minimal and maximal times hold absolute times (e.g. 14/11/2011 13:26), not relative (e.g. 3 hours). During the run of the CreateTimePoints function, the ZeroTime has been determined appropriately and the current times were set according to the temporal data of the nodes in the input schedule. The minimal (maximal) times have not been treated sufficiently yet. They have been set to the ZeroTime (resp. the maximal allowed time in the resulting schedule), but they do not reflect constraints introduced to the STP graph in the ConvertConstraints function. So, now it is time to resolve this inconsistence.

If we take a look at the Figure 44, we can see that there are relative times introduced among the time points. Since the IFPC algorithm (see Incremental Full Path Consistency (IFPC) algorithm) guarantees the property *all-pairs shortest path* of the resulting STP graph, the relative times are propagated to constraints

$Dist(the\ zero\ point, other\ point\ P) = [min, max]$. Note that all *min* bounds are determined since the zero point is the global predecessor of all the other points and the zero point holds the only absolute time (i.e. ZeroTime) which is always valid. The *max* bounds need not to be determined because there is no global successor of all time points as the zero time is the global predecessor. More precisely, the *max* bound is set by some points if the input schedule contains pins. We can notice that *min* (*max*) actually represents the new value of the minimal (maximal) time of the point *P* if it is more restrictive, i.e. greater (smaller), than the current value of the minimal (maximal) time. If the current time of the point *P* is out of range of the updated bounds (= the new minimal and maximal times), it is set to the value of the nearer bound ("nearer" in time). It follows that this movement of *P* will be reflected on the rating of the resulting schedule which cannot equal zero anymore because of that.

The implementation of the UpdateTimes function is quite straightforward; hence we demonstrate the function on the sample schedule depicted in the Figure 40 and Figure 41. The following Figure 45 extends the Figure 44.

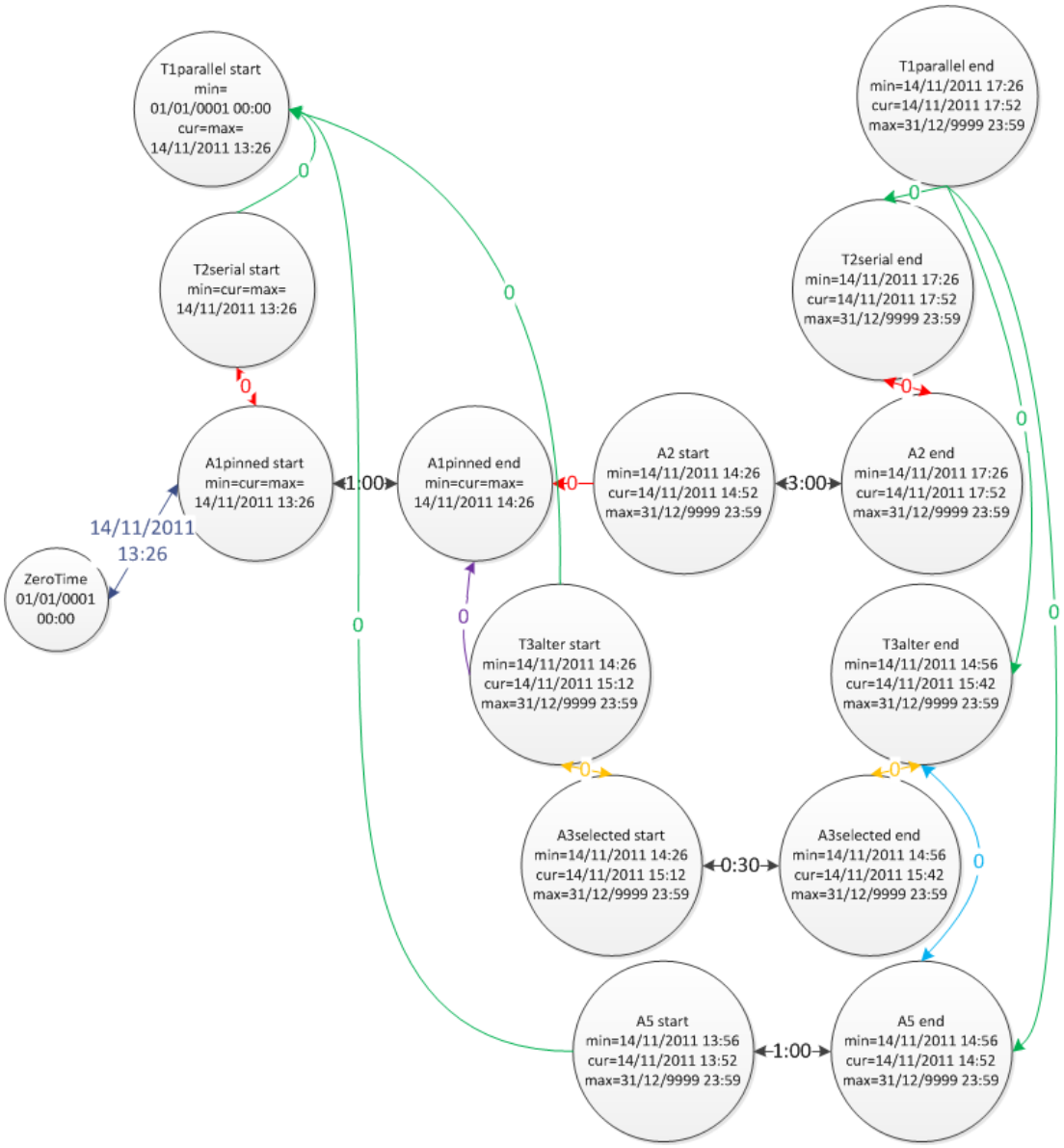


Figure 45: The state of the STP graph after a computation of the minimal and maximal bounds, but before the update of the current times (see the time points of the activity A5). *min* (*cur*, *max*) property by a particular time point means the minimal (current, maximal) time of that point.

We should notice four things in Figure 45:

- Pin – The pin on the activity A1pinned does not pin only two time points of the activity, but also the time point *T2serial start*.
- Maximal times – Since there is no global successor, the maximal time of for instance *A2 start* is 31/12/9999 23:59, not 31/12/9999 20:59. The reason is that no arrow from the zero point to *A2 start*, i.e. no constraint like $(A2\ start - the\ zero\ point \leq max)$, exists and the IFPC algorithm does not add such a constraint.
- Arrows – The STP graph in the figure contains only arrows added there directly by the ConvertConstraints function, though there should be

precedencies $(the\ zero\ point \rightarrow other\ point)$ and arrows added by the IFPC algorithm.

- Reservations – We should keep in mind that the current state of the STP graph reflects only simple constraints. There are no constraints of disjunctions. That is why for instance the minimal time of the time point *T1parallel start* equals the ZeroTime, though it is evident from the minimal times of the time points of the task's children (T2serial start, T3alter start, A5 start) that the minimal time of *T1parallel start* is 14/11/2011 13:26. This situation and the similar ones are discussed in the chapter Pruning of Disjunctions later on.

UpdateTimes function – Proofs of Completeness and Soundness

The soundness of the UpdateTimes function is evident from the description and the sample above. The complexity is $O(n)$ where *n* is the number of all time points since the IFPC algorithm secures that it is sufficient to go through all constraints which point to the zero point and come from the zero point. The reason is the property *all-pairs shortest path* (i.e. propagation of constraints).

8.6 Solving of DTP

Till now we have created an STP graph *G* from simple constraints, adjusted temporal data of time points according to these constraints and produced disjunctions (the list of such disjunctions named *DI*). The aim of this – final – step (the steps 5-8 in the sketch of the Repair-DTP algorithm) covered by the SolveDTP function is to explore all combinations of the constraints of the unresolved disjunctions *DI* and try to add these combinations to the graph *G*. If we get a consistent graph *G'* which satisfies all of these disjunctions *DI*, the minimal times of all time points in the obtained graph *G'* represent *the feasible minimal schedule (MS)*; it follows from the existence of the global predecessor of all time points (except the zero point) anchored in time and the usage of the IFPC algorithm. Nevertheless, we would like to get the most similar schedule to the input infeasible one. Therefore, we call a modification of the Precedence Repair (PredRep) algorithm with the found minimal schedule (MS) as a parameter. We evaluate *the found current schedule* (CS) according to the function f_1 (see Evaluation/Rating function) and if the rating is better than the best rating computed till that time, we save the found current schedule (CS). At the end of the SolveDTP function, we return the found current schedule (CS) with the best rating.

SOLVEDTP

Input: DTP problem *dtp* with STP graph *G* and disjunctions *DI*, *disjIndex* – number of disjunction in *DI* which is to be processed

Output: *dtp* with the best solution if the solution exists

```
1   D ⟵ disjunction of DI with the number disjIndex
2   FOR EACH C ∈ Constraints(D)
3      IF IsConsistent(C, G) ≠ INCONSISTENT
4         add C to G
5
6         IF disjIndex = |DI|
7            solution ⟵ FindCurrent(G)
8            IF ∄BestSolution(dtp) OR f1(solution) < f1(BestSolution(dtp))
9               BestSolution(dtp) ⟵ solution
10           END IF
11           rollback "add C to G",UpdateTimes(G),FindCurrent(G)
12        ELSE
13           SolveDTP(dtp, disjIndex + 1)
14        END IF
15     END IF
16  END FOR
17  return dtp
```

Figure 46: The SolveDTP function of the Repair-DTP algorithm

In the Repair function (see Skeleton of Repair-DTP), the SolveDTP function is called with two parameters: *dtp* which represents the whole scheduling problem and the parameter *disjIndex* = 1. The parameter *disjIndex* determines the disjunction *D* (disjunctions *DI* are numbered) which is to be processed in the current call of the SolveDTP function (line 1). It follows that the SolveDTP function has a recursive implementation (line 13).

The SolveDTP function iterates successively through all constraints of the disjunction *D* (line 2) till the time, the function finds a constraint *C* which is consistent with the current STP graph *G* (line 3). Then, the function adds the constraint *C* to the graph *G* via the IFPC algorithm and immediately recomputed affected minimal, maximal and current times. If we have already found such a constraint *C* for each disjunction from *DI* (line 6), it means that we have found a desired combination of constraints which is consistent with the graph *G*. Then, the graph *G* represents one of *the feasible minimal schedules (MS)* since the minimal times of all time points satisfy all disjunctions in the input infeasible schedule.

As we proposed, we call the function *FindCurrent* (presented later on) which finds an appropriate *feasible current schedule* (CS) called *solution* for the current minimal schedule (MS) *G* (line 7). When the FindCurrent function ends, we find out how good the current schedule (CS) is (line 8) and if the schedule is better than the best one found till that time, we save the found schedule (line 9).

Since more consistent combinations of constraints of disjunctions *DI* can exist and we want to explore all of them, we rollback the operations related to the last added constraint (line 11) when we finish processing of the found feasible minimal

schedule (MS). Afterwards, we continue with exploration of the remaining combinations.

We continue with the sample depicted in the Figure 45. The following Figure 47 shows a feasible minimal schedule (MS) where the emphasized (in bold) constraints of disjunctions *DI* are satisfied:

1. $\boldsymbol{Dist}\big(\boldsymbol{Start}(\boldsymbol{T1parallel}), \boldsymbol{Start}(\boldsymbol{T2serial})\big) = [\mathbf{0}, \mathbf{0}] \vee$
   $Dist\big(Start(T1parallel), Start(T3alter)\big) = [0,0] \vee$
   $Dist\big(Start(T1parallel), Start(A5)\big) = [0,0]$
2. $\boldsymbol{Dist}\big(\boldsymbol{End}(\boldsymbol{T1parallel}), \boldsymbol{End}(\boldsymbol{T2serial})\big) = [\mathbf{0}, \mathbf{0}] \vee$
   $Dist\big(End(T1parallel), End(T3alter)\big) = [0,0] \vee$
   $Dist\big(End(T1parallel), End(A5)\big) = [0,0]$
3. $\boldsymbol{Dist}\big(\boldsymbol{End}(\boldsymbol{A1pinned}), \boldsymbol{Start}(\boldsymbol{A5})\big) = [\mathbf{0}, \infty] \vee$
   $Dist\big(End(A5), Start(A1pinned)\big) = [0, \infty]$

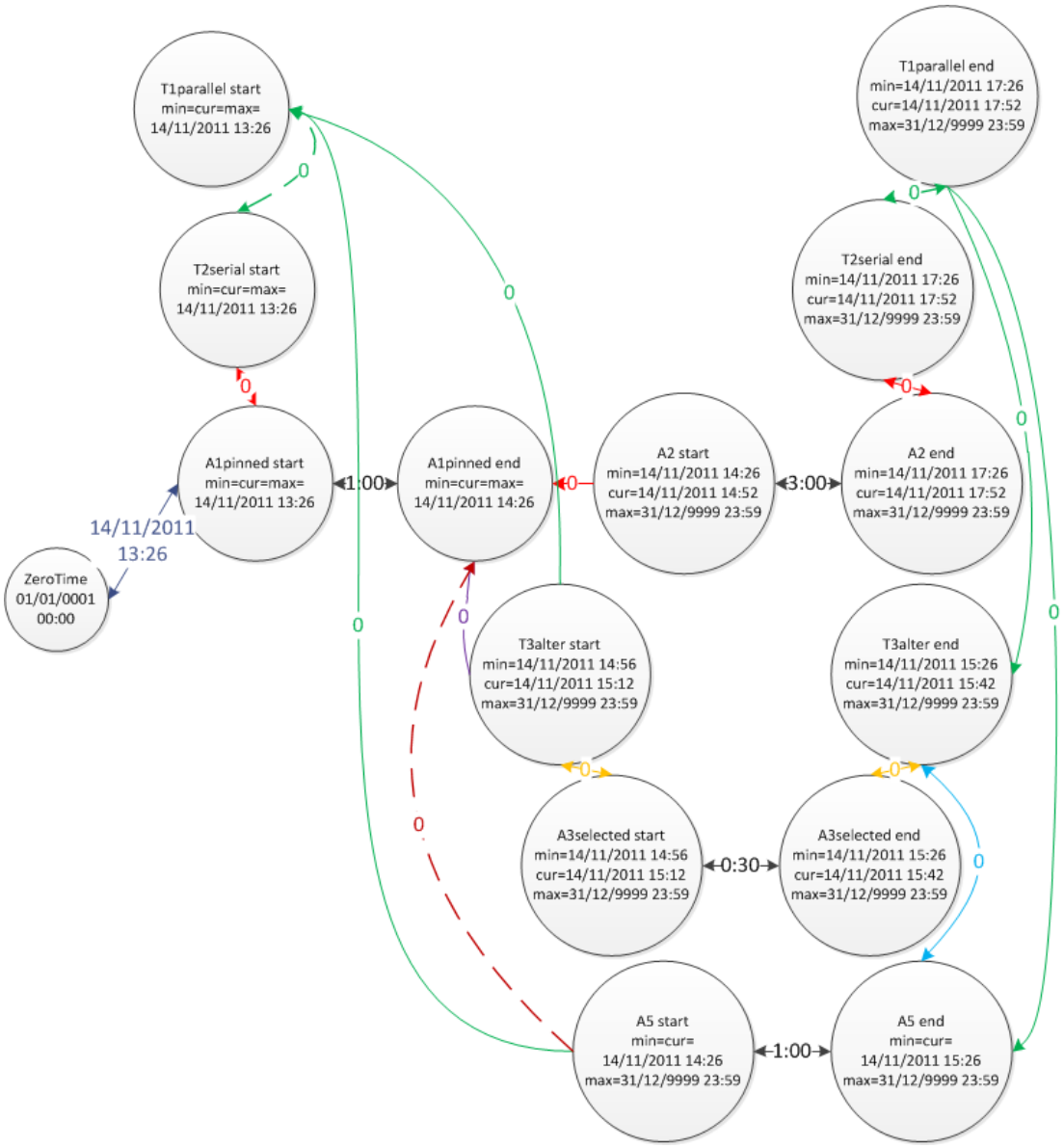


Figure 47: The feasible minimal schedule (MS) which satisfies all constraints including the dashed ones.

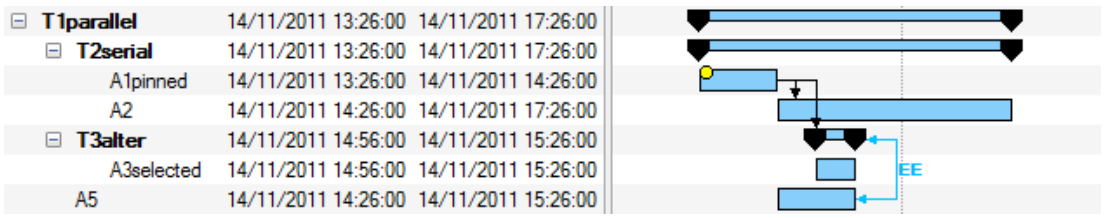


Figure 48: The feasible minimal schedule (MS) – the Gantt Chart

In the STP graph in Figure 47, we can see that the current times of all time points do not create a feasible current schedule (CS) since the constraint $Dist\big(CurrentTime(T3alter\ end), CurrentTime(A5\ end)\big) = [0,0]$ is violated.

Functions

There are two functions in the pseudo code of the SolveDTP function which have not been discussed in detail yet: *IsConsistent*, *FindCurrent*.

IsConsistent function

The whole implementation of the IsConsistent function is actually one line – the first line of the pseudo code of the IFPC algorithm (see Figure 35). It decides in the constant time whether a constraint *C* is inconsistent with the current STP graph, or consistent. It does not matter whether the constraint *C* is redundant with respect to the graph, or not.

In the following chapters we will familiarize with the FindCurrent function. For the proofs of the completeness and soundness of the SolveDTP function, see SolveDTP function – Proofs of Completeness and Soundness.

8.6.1 Find the Feasible Current Schedule (CS)

This chapter discusses the function FindCurrent which gets a STP graph *G* with the feasible minimal schedule (MS) and returns the graph with the feasible current schedule (CS) and the input minimal schedule (MS) as well. In follows that the FindCurrent function has to resolve all constraint violations in the input current schedule (CS) and this resolving is done with the respect to the rating of the resulting feasible current schedule (CS).

FINDCURRENT

Input: STP graph *G* with the feasible minimal schedule (MS)

Output: *G* with both the feasible minimal and current schedule (MS, CS)

1 $violated \leftarrow \{C | C \in \text{Constraints}(G) \wedge \text{IsCurrentViolated}(C)\}$
2 WHILE $violated \neq \emptyset$
3 $C \leftarrow$ get constraint from *violated*
4 remove *C* from *violated*
5 RepairCurrentConstraint(*C*)
6
7 $A_C \leftarrow \text{From}(C); B_C \leftarrow \text{To}(C)$
8 $violated \leftarrow \{(A,B) | (A,B) \in violated \wedge A \notin \{A_C, B_C\} \wedge B \notin \{A_C, B_C\}\} \cup$
9 $\{(A,B) | (A,B) \in \text{Constraints}(G) \wedge \text{IsCurrentViolated}(C) \wedge (A \in \{A_C, B_C\} \vee B \in \{A_C, B_C\})\}$
10 END WHILE
11 return *G*

Figure 49: The FindCurrent function of the Repair-DTP algorithm

Since the FindCurrent function has to correct all violated constraints, it finds all such constraints in the current schedule (CS) of the input STP graph *G* (line 1) and the function loops until there is a violated constraint in the current schedule (CS) of the current *G* (line 2). In each iteration of the loop, one violated constraint $C = (A_C, B_C)$ is repaired (line 3-5). However, correction of *C* can cause that some other constraints $(A_C, X), (X, A_C), (B_C, X), (X, B_C), X \in TimePoints(G)$ are either repaired as well, or get violated. Therefore, violated constraints which are not connected with neither A_C, nor B_C are left in the variable *violated* (line 8), and all the others are explored from scratch (line 9).

Functions

Before we prove the completeness and soundness of the FindCurrent function (see FindCurrent function – Proofs of Completeness and Soundness) we should focus on the functions *IsCurrentViolated* and *RepairCurrentConstraint* used in the pseudo code above.

IsCurrentViolated function

If there is a constraint *C* between two time points *A* and *B*, the weight *w* of the constraint *C* determines the upper bound of the time distance between the times of these time points. Since the minimal schedule (MS) in the STP graph *G* is feasible, it stands for each constraint *C* that $(MinTime(B) - MinTime(A) \leq w)$. However, it may happen that $(CurrentTime(B) - CurrentTime(A) = w' > w)$ (see Figure 47). In that case, the constraint *C* is violated in the current schedule (CS) and needs to be repaired. Note that the decision whether a particular constraint is violated or not can be done in the constant time.

We should add that $(MinTime(B) - MinTime(A) = w)$ since if the distance is smaller, i.e. $(MinTime(B) - MinTime(A) < w)$, one of the times is not minimal.

RepairCurrentConstraint function

The function which is left for our discussion is RepairCurrentConstraint. The function provides a correction of a violated constraint $C = \left(A \overset{w}{\rightarrow} B\right)$ of the STP graph *G*, i.e. the correction of the constraint *C* where $(CurrentTime(B) - CurrentTime(A) = w' > w)$.

REPAIRCURRENTCONSTRAINT

Input: violated constraint *C* = $\left(A \overset{w}{\rightarrow} B\right)$ of the STP graph *G*

Output: corrected *C*

```
1   overflow ← CurrentTime(B) – CurrentTime(A) – w
2   CurrentTime(A) ← CurrentTime(A) + ⌊overflow/2⌋
3   CurrentTime(B) ← CurrentTime(B) – ⌈overflow/2⌉
4   return C
```

Figure 50: The RepairCurrentConstraint function of the Repair-DTP algorithm

Note that the first line of the pseudo code above is a reformulation of a condition in the IsCurrentViolated function. So, the variable *overflow* determines about how much time the time distance between *CurrentTime(A)* and *CurrentTime(B)* must be reduced. The following two lines contain a shifting of the current times of *A* and *B* and it is important that the sum of both times decreases, or stays the same, but not increases. This fact guarantees us that we always find the feasible current schedule in the FindCurrent function as we will see later on.

RepairCurrentConstraint function – Proofs of Completeness and Soundness

First we prove the correctness. On the line 1 of the pseudo code, there is an equation:

$$overflow = CurrentTime(B) - CurrentTime(A) - w$$

After the lines 2 and 3 have been performed, the same equation should look:

$$0 = CurrentTime(B)_{new} - CurrentTime(A)_{new} - w$$

where *new* indicates the new value of the current time of a particular time point.

$$0 = \left(CurrentTime(B) - \left\lceil\frac{overflow}{2}\right\rceil\right) - \left(CurrentTime(A) + \left\lfloor\frac{overflow}{2}\right\rfloor\right) - w$$

$$0 = CurrentTime(B) - CurrentTime(A) - w - \left\lceil\frac{overflow}{2}\right\rceil - \left\lfloor\frac{overflow}{2}\right\rfloor$$

$$overflow = CurrentTime(B) - CurrentTime(A) - w$$

The soundness of the RepairCurrentConstraint function has been hence proved. The complexity of the function is $O(1)$ since all operations in the pseudo code are performed in the constant time.

FindCurrent function – Proofs of Completeness and Soundness

Now, we can focus on the proofs related to the FindCurrent function. Since the function ends just in case there is no violated constraint in the current schedule (CS) of the STP graph *G* and the called functions IsCurrentViolated and RepairCurrentConstraint do what they should do, the FindCurrent function is sound.

In terms of the completeness of the function, we need to prove two lemmas first:

Lemma 1: Assume that *p* is a time point different from the zero one with a valid current time, i.e. MinTime(*p*) ≤ CurrentTime(*p*) ≤ MaxTime(*p*). Then, the current time of *p* stays valid during the whole execution of the FindCurrent function.

Proof of Lemma1: The current time of the time point *p* can be modified only in the RepairCurrentConstraint function and just in cases when $\exists C = \left(A \overset{w}{\rightarrow} B\right), p \in \{A, B\}$ and IsCurrentViolated(*C*) = true. It follows:

$$(1) \quad CurrentTime(B) - CurrentTime(A) - w = \boldsymbol{overflow} > \mathbf{0}$$

That is exactly the content of the line 1 in the RepairCurrentConstraint function. Recall that *C* is a constraint of the STP graph *G*, not of the input infeasible schedule.

The constraint *C* can be one of two types:

1. The constraint *C* determines the *minimal* time distance between *A* and *B*, i.e.

$$-w_{(B \rightarrow A)} \leq CurrentTime(B) - CurrentTime(A)$$

$$(2) \quad CurrentTime(A) - CurrentTime(B) \leq w_{(B \rightarrow A)}$$

2. The constraint *C* determines the *maximal* time distance between *A* and *B*, i.e.

$$(3) \quad CurrentTime(B) - CurrentTime(A) \leq w_{(A \rightarrow B)}$$

Note that $w_{(B\to A)} \leq 0$ in the equation (2), but $w_{(A\to B)} \geq 0$ in the equation (3). In both (2) and (3) times of the point *B* are greater than *A*'s times, i.e. *B* is more to the right than *A*.

It follows that there are two parts of the proof of the lemma 1: one part which considers the minimal times and the other which considers the maximal times.

Now we focus on the minimal times. First of all, we use the computation of the minimal times from the UpdateTimes function. Thanks to the usage of the zero time point which is connected to all the other points with precedencies $(the\ zero\ point \to other\ point)$ and the usage of the IFPC algorithm which secures that constraints which are being added to the STP graph *G* are propagated, we can compute the minimal times of all time points (except the zero point). We need only the ZeroTime and the weights of the precedencies $(the\ zero\ point \to other\ point)$, i.e. weights of the constraints $(other\ point \to the\ zero\ point)$.

$$(4) \quad MinTime(B) = ZeroTime - w_{(B\to zero)}$$

The value $w_{(B\to zero)}$ represents the weight of the constraint $\left(B - the\ zero\ point \geq -w_{(B\to zero)}\right)$. It follows that $w_{(B\to zero)} \leq 0$. Thus, thanks to the propagation of constraints provided by the IFPC algorithm:

$$(5) \quad MinTime(A) - MinTime(B) =$$
$$ZeroTime - w_{(A\to zero)} - \left(ZeroTime - w_{(B\to zero)}\right) =$$
$$-\left(w_{(A\to zero)} - w_{(B\to zero)}\right) = w_{(B\to A)}$$

Since the weight $w_{(B\to A)}$ restricts the time distance between two minimal times, the weight makes a sense only when its values is less or equal 0 (see the discussion at the beginning of the proof). Further, we should say that all weights in the whole proof are determined by the propagation of constraints provided by the IFPC algorithm, not by the differences between current times of time points, i.e. *w* in the equation (1), $w_{(B\to A)}$ in (2), $w_{(A\to B)}$ in (3), $w_{(B\to zero)}$ in (4) and so on are constant during the finding a feasible current schedule (CS). Now, let us take the inequality (2):

$$CurrentTime(A) - CurrentTime(B) \leq w_{(B\to A)}$$

$$(6) \quad CurrentTime(A) - CurrentTime(B) - w_{(B\to A)} \leq 0$$

The inequality (6) determines the valid values of the current times of *A* and *B*. If the left side of (6) is greater than 0, we get a violated constraint $(B \to A)$:

$$(7) \quad \boldsymbol{CurrentTime(A) - CurrentTime(B) - w_{(B\to A)} = overflow > 0}$$

That is exactly the equality (1). We can substitute the weight in (7) by the left side of the equality (5):

$$CurrentTime(A) - CurrentTime(B) - \left(MinTime(A) - MinTime(B)\right) = overflow > 0$$

$$CurrentTime(A) - MinTime(A) - \big(CurrentTime(B) - MinTime(B)\big) = overflow > 0$$

Since the time is discrete in the FlowOpt project (the minimal time unit is a minute):

(8a) $$CurrentTime(A) - MinTime(A) - \big(CurrentTime(B) - MinTime(B)\big) = overflow \geq 1$$

(8b) $$CurrentTime(A) - MinTime(A) - \big(CurrentTime(B) - MinTime(B)\big) \geq 1$$

Statement $CurrentTime(B) \geq MinTime(B)$ always stands because the point B is always moved to the right during the repair of violations of the first type; hence:

(9a) $$CurrentTime(B) - MinTime(B) \geq 0$$

(9b) $$-\big(CurrentTime(B) - MinTime(B)\big) \leq 0$$

As a result of (8b) and (9b):

$$CurrentTime(A) - MinTime(A) \geq 1$$

(10) $$CurrentTime(A) \geq MinTime(A) + 1$$

Now, we have to show that the point A, which is always moved to the left during the repair of violations of the first type (i.e. the minimal distances), does never violate the statement $CurrentTime(A) \geq MinTime(A)$.

In the RepairCurrentConstraint function on the line 3:

$$CurrentTime(A)_{new} = CurrentTime(A) - \left\lceil \frac{overflow}{2} \right\rceil = CurrentTime(A) - \left\lfloor \frac{overflow + 1}{2} \right\rfloor = \left\lfloor \frac{2 * CurrentTime(A) - overflow - 1}{2} \right\rfloor$$

If we substitute the variable *overflow* with the equation (8a):

$$\left\lfloor \frac{2 * CurrentTime(A) - \big(CurrentTime(A) - MinTime(A) - \big(CurrentTime(B) - MinTime(B)\big)\big) - 1}{2} \right\rfloor$$

=

(12) $$\left\lfloor \frac{CurrentTime(A) + MinTime(A) + \big(CurrentTime(B) - MinTime(B)\big) - 1}{2} \right\rfloor$$

When we apply the inequality (9a):

(13) $$(12) \geq \left\lfloor \frac{CurrentTime(A) + MinTime(A) - 1}{2} \right\rfloor$$

And the inequality (10):

$$(13) \geq \frac{2 * MinTime(A)}{2} = MinTime(A)$$

As it turned out, the new current time of the point *A* is never smaller than the *A*'s minimal time. So, we have proved that the lemma 1 stands when the constraint *C* determines the minimal distance between *A* and *B*, i.e. is of the first type.

The proof of the second part of the lemma 1 is analogous. Though we go through it shortly.

So, we consider now that the constraint *C* determines the maximal distance between *A* and *B*:

$$(3) \quad CurrentTime(B) - CurrentTime(A) \leq w_{(A \to B)}$$

If the constraint *C* is violated, *B* is moved to the left and *A* to the right in order to correct the violation. It follows that we need to prove that *A* does not cross its maximal time. By the point *B*, there is nothing to prove. We should keep in mind that there is no global successor, as the zero point is the global predecessor; hence if the point *A* has not its maximal time set, no proof is necessary and the lemma 1 stands.

Assume that the *A*'s maximal time is set. It means:

$$(14) \quad MaxTime(A) = w_{(zero \to A)} - ZeroTime$$

Thanks to the *A*'s maximal time, i.e. the constraint $(the\ zero\ point \to A)$, and since there is the maximal distance between the points *A* and *B*, i.e. the constraint $(A \to B)$, and the IFPC algorithm secures the propagation of constraint, we get that the point *B* has also the maximal time, i.e. the constraint $(the\ zero\ point \to B)$. Then:

$$(15) \quad MaxTime(B) - MaxTime(A) = \\ w_{(zero \to B)} - ZeroTime - \left(w_{(zero \to A)} - ZeroTime\right) = \\ w_{(zero \to B)} - w_{(zero \to A)} = w_{(A \to B)}$$

If we take (3) and we want to express a violated constraint, we get (in)equality similar to (1) and (7):

$$(\mathbf{16}) \quad \boldsymbol{CurrentTime(B) - CurrentTime(A) - w_{(A \to B)} = overflow > 0}$$

We substitute the weight $w_{(A \to B)}$ with the left side of (15):

$$CurrentTime(B) - CurrentTime(A) - \left(MaxTime(B) - MaxTime(A)\right) = overflow > 0$$

$$(17) \quad CurrentTime(B) - MaxTime(B) + \left(-CurrentTime(A) + MaxTime(A)\right) = overflow > 0$$

We know that $CurrentTime(B) \leq MaxTime(B)$ always stands:

$$(18) \quad CurrentTime(B) - MaxTime(B) \leq 0$$

We combine (17) and (18):

$$-CurrentTime(A) + MaxTime(A) > 0$$

$$(19) \quad CurrentTime(A) < MaxTime(A)$$

In the RepairCurrentConstraint function on the line 2:

$$CurrentTime(A)_{new} = CurrentTime(A) + \left\lfloor \frac{overflow}{2} \right\rfloor = \left\lfloor \frac{2 * CurrentTime(A) + overflow}{2} \right\rfloor$$

If we substitute the variable *overflow* with the equation (17):

$$\left\lfloor \frac{2 * CurrentTime(A) + CurrentTime(B) - MaxTime(B) + \left(-CurrentTime(A) + MaxTime(A)\right)}{2} \right\rfloor$$

$$=$$

$$(20) \quad \left\lfloor \frac{CurrentTime(B) - MaxTime(B) + \left(CurrentTime(A) + MaxTime(A)\right)}{2} \right\rfloor$$

With using the inequality (18):

$$(21) \quad (20) \leq \left\lfloor \frac{CurrentTime(A) + MaxTime(A)}{2} \right\rfloor$$

Finally we apply (19):

$$(21) \leq \left\lfloor \frac{2 * MaxTime(A)}{2} \right\rfloor = MaxTime(A)$$

We have successfully proved that the new current time of *A* will never be greater than *A*'s maximum time. Thus, the lemma 1 stands.

Lemma 2: When the RepairCurrentConstraint function runs, the total sum of the current times of all time points either decreases, or stays the same and no constraint is repaired by the function twice at the same time.

Proof of Lemma 2: This lemma has not been successfully proved.

Proposition 1: The FindCurrent function is complete.

Proof of Proposition 1: First of all we should say that this proof is based on the validity of the lemmas 1 and 2. Since the latter has not been correctly proved, the proof of the proposition 1 which follows is only a sketch. However, if the lemma 2 was proved, the following text would be the proper proof as well.

Assume the constraint $C = \left(A \overset{w}{\rightarrow} B\right)$ is to be repaired in the RepairCurrentConstraint function. Since its *overflow* ≥ 0 (line 1 in the RepairCurrentConstraint function) and

the upper bound $\left\lceil \frac{overflow}{2} \right\rceil$ is subtracted (line 3), not added, the RepairCurrentConstraint function shifts *CurrentTime(B)* more or equal to the left, as *CurrentTime(A)* to the right. It follows that each iteration of the FindCurrent function decreases the total sum of the current times of all time points (we denote it ∑), or does not change the sum ∑.

If we proved the lemma 2, we would know that we could encounter two situations concerning the sum ∑: (1) either no constraint in the STP graph would be repaired twice, or (2) the sum ∑ would decrease. In the case (1), there would be at most *m* iterations of the while-loop where *m* would be the number of constraints in the graph. Otherwise, some constraints would be repaired more times.

Thanks to the lemma 1 we know that we cannot cross the minimal time of none of the time points with the new current time. Further, the minimal times of all points represent the feasible minimal schedule (MS) in the FindCurrent function and they are constant during the run of the FindCurrent function. So, if the lemma 2 was proved: we would know that after the finite number of iterations of the while-loop we would get the feasible current schedule (CS) since the feasible minimal schedule (MS) exists and the sum ∑ would decrease (the second case). This current schedule (CS) would be either the same as the minimal schedule (MS), or the current schedule would be more similar to the input infeasible schedule.

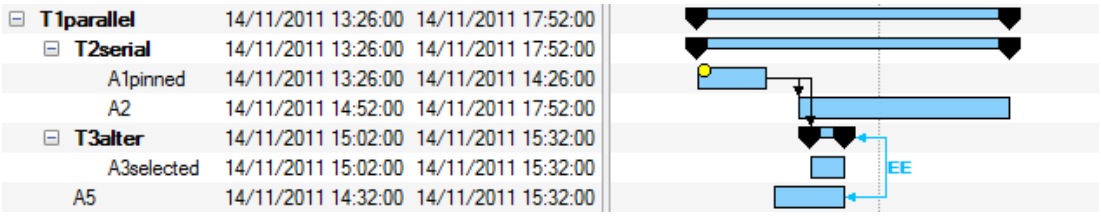


Figure 51: Feasible current schedule (CS) for the minimal schedule (MS) in Figure 47. Note that the former (f_1 = 50 minutes) is more similar to the input schedule than the latter (f_1 = 76 minutes). The activity A5 starts six minutes later.

8.6.2 SolveDTP function – Proofs of Completeness and Soundness

The soundness of the SolveDTP function has been shown by the description below the code of the function (see Solving of DTP). The proof of completeness is partly done. It means that all functions and operations in the SolveDTP function (i.e. IsConsistent, "add *C* to *G*", UpdateTimes, f_1 and "rollback…") except the FindCurrent function have been proved to be complete and there is only the finite number of the combinations of the constraints of the disjunctions *DI*. If the lemma 2 is valid, the SolveDTP function will be complete as well.

Number of iterations

The recursion in the SolveDTP function and the for-loop over all constraints in the disjunctions *DI* imply that the total number of iterations of the loop in all calls of the SolveDTP function should be very high. Nevertheless, we are able to find the upper

bound of the number of these iterations. Since the disjunctions *DI* come from parallel tasks and reservations (see Disjunctive Temporal Problem (DTP)):

$$Max(|DI|) = Max(|disjunctions\ of\ DI\ coming\ from\ parallel\ tasks|) + Max(|disjunctions\ of\ DI\ coming\ from\ reservations|)$$

We shorten *disjunctions of DI coming from parallel tasks* to $DI_{parallel}$ and *disjunctions of DI coming from reservations* to $DI_{reservation}$. Then,

$$(21) \quad Max(|DI|) = Max(|DI_{parallel}|) + Max(|DI_{reservation}|)$$

For each parallel task there can be at most two disjunctions of $DI_{parallel}$. The maximal number of disjunctions $DI_{reservation}$ related to a resource *R* equals $\binom{n}{2}$ where *n* is the total number of the scheduled activities in the schedule. We get $\binom{n}{2}$ when all scheduled activities allocate the resource *R*. When all scheduled activities allocate all available resources (the total number of the resources is *m*), we get:

$$(21) = 2 * Max(|parallel\ tasks|) + m * \binom{n}{2}$$

Assume that each parallel task has at least two children; otherwise, its disjunctions would have only one constraint (i.e. they are simple constraints actually) and these types of disjunctions are pruned when an appropriate optimization is used (see Pruning of Disjunctions). Then, we get the maximal number of the parallel tasks when all tasks in the schedule are parallel and each of them has exactly two children. Then, the number of the parallel tasks in the schedule is:

$$1 + 2 + 4 + 8 \ldots \frac{n}{2}$$

where 1 is for the root task, 2 for root's direct children etc. The number of the summands is $\log_2 n$. When we use a formula for the sum of the geometric series $a_1 \frac{q^k - 1}{q - 1}$, we get that the maximal number of the parallel tasks in the schedule is (n – 1):

$$(21) = 2 * (n - 1) + m * \binom{n}{2}$$

We continue with the computation of the maximal number of combinations of constraints in the disjunctions *DI*. Note that this number equals the maximal number of iterations of the for-loop in the SolveDTP function. Assume that each disjunction *D* in *DI* had exactly two constraints. So, either the first constraint of *D* can be selected, or the second constraint. Then, we get that the maximal number of combinations would be $2^{|DI|}$. We apply this idea on our problem:

$$(22) \quad Max(|iterations|) =$$
$$Max\left(\left|constraints\ for\ one\ of\ DI_{parallel}\right|\right)^{Max(|DI_{parallel}|)}$$
$$* Max(|constraints\ for\ one\ of\ DI_{reservation}|)^{Max(|DI_{reservation}|)} =$$
$$Max\left(\left|constraints\ for\ one\ of\ DI_{parallel}\right|\right)^{(2*(n-1))}$$
$$* Max(|constraints\ for\ one\ of\ DI_{reservation}|)^{\left(m*\binom{n}{2}\right)}$$

Each disjunction of $DI_{reservation}$ has always two constraints. The number of constrains in disjunctions of $DI_{parallel}$ is at most *n* where *n* is the total number of activities in the schedule since each task in the schedule has to have at least one child. We get:

$$(22) = n^{(2*(n-1))} * 2^{\left(m*\binom{n}{2}\right)} \approx n^n * 2^{mn^2}$$

Right now we showed that the temporal complexity of the SolveDTP function is exponential.

8.7 Complexity

There are two perspectives we should consider in terms of complexity. The first view is the temporal complexity and the other is the spatial complexity. In the previous chapter we have shown that the temporal complexity is determined by the SolveDTP function and is exponential. In the worst case, the complexity is $O\left(n^n * 2^{mn^2}\right)$ where *n* is the total number of the scheduled activities in the input infeasible schedule and *m* is the total number of available resources.

The spatial complexity of the presented algorithm is also not so favorable. The reason is that the algorithm creates two time points for each selected node; plus the zero point. Afterwards, it is necessary to convert all – both implicit and explicit – constraints which are in the schedule to an appropriate STP graph *G*; hence the maximal number of constraints in *G* is $(2 * |SelectedNodes| + 1)^2$. Thus, for 500 selected nodes there are more than million constraints. In the chapter Experiments the real impact on the algorithm's requirements is demonstrated.

8.8 Similarity

Now we should discuss the similarity of the resulting feasible schedule to the input infeasible one. As we have already implied, the Repair-DTP algorithm does not guarantee that the resulting schedule is the most similar feasible schedule to the input one according to the evaluation function f_1. There are more reasons. The first one is that if the input schedule does not contain a pin, the zero time point is set to the start time of the earliest scheduled activity in the input schedule; hence all potential feasible schedules which would start before this time cannot be found by the algorithm.

Even if the input schedule contains at least one pin, the search space is pruned by the RepairCurrentConstraint function. The function gets a violated constraint *C* and repairs it. In other words, the function transforms the STP graph *G* (with the violated *C*) to *G'* (without the violated *C*) and it stands:

$$\sum_{p \in Points(G_F)} CurrentTime(p) \leq \sum_{p \in Points(G')} CurrentTime(p)$$
$$\leq \sum_{p \in Points(G)} CurrentTime(p) \leq \sum_{p \in Points(G_S)} CurrentTime(p)$$

where G_S is the STP graph with a feasible minimal schedule (MS) which is the input parameter of the FindCurrent function; G_S is the corresponding feasible current schedule (CS) which is returned by the function. The inequality above follows from the proofs of the FindCurrent function. We can notice that the G_F's sum can never be greater than G_S's sum; hence the search space is pruned.

In total, though the RepairCurrentConstraint function uses a promising approach of drawing time points near, the whole Repaid-DTP algorithm cannot guarantee that the schedule which the algorithm returns is the best feasible schedule which exists.

8.9 Optimizations

In the previous chapters we have familiarized with the core of the Repair-DTP algorithm. Here, we introduce three optimizations of the Repair-DTP algorithm which improve the algorithm's performance.

8.9.1 Pruning of Disjunctions

In the ConvertToDTP function, the DTP problem which represents the original infeasible schedule is created. The problem consists of a STP graph of the simple constraints and disjunctions *DI*. However, when the simple constraints are being added to the graph, ranges of valid values of the current times of time points are being restricted more and more. Therefore, when the ConvertToDTP function ends and returns the DTP problem, some constraints in the disjunctions *DI* may not be consistent with the STP graph anymore and some may be redundant; hence we do a pruning of the disjunctions *DI*. As a result, some constraints of disjunctions *DI* or even the whole disjunctions may be removed.

PRUNEDISJUNCTIONS
Input: DTP problem *dtp* with STP graph *G* and disjunctions *DI*
Output: *dtp* with extended *G* and pruned *DI*

```
1   REPEAT
2      graphModified ⟵ FALSE
3      FOR EACH D ∈ DI
4         FOR EACH C ∈ Constraints(D)
5            IF IsConsistent(C, G) = INCONSISTENT
6               remove C from D
7            ELSE IF IsConsistent(C, G) = REDUNDANT
8               remove D from DI
9               break both loops
10           END IF
11        END FOR
12
13        IF |Constraints(D)| = 0
14           FAIL
15        IF |Constraints(D)| = 1
16           add C to G
17           remove D from DI
18           graphModified ⟵ TRUE
19        END IF
20     END FOR
21  UNTIL graphModified = FALSE
22  return dtp
```

Figure 52: The PruneDisjunctions function of the Repair-DTP algorithm

The *PruneDisjunctions* function goes through all constraints of all disjunctions *DI* (lines 3 and 4) and checks whether a particular constraint *C* of a disjunction *D* is inconsistent with the current STP graph, or redundant (this pruning described in [13]). If *C* is inconsistent, the constraint is eliminated from the disjunction *D* (line 6); if the constraint *C* is redundant, it means that the disjunction *D* is always satisfied and hence we can eliminate *D* at all (line 8) and continue with the checking of the next disjunction *DI* (line 9).

When we have already checked all constraints of the disjunction *D*, we have to find out how many constraints are left in *D*. If none is left (line 14), the disjunction *D* cannot be satisfied and hence no feasible schedule for the input one exists and the Repair-DTP algorithm fails. If more than one constraint is left, the disjunction *D* cannot be still resolved unambiguously; thus *D* stays in *DI*. The last possible case is that exactly one constraint *C'* is left in the disjunction *D* (line 15). It means that *D* is satisfied when *C'* is; hence we add *C'* to the graph and remove *D* from *DI* (lines 16-17). However, since the constraint *C'* has not been redundant, the adding of *C'* to the graph causes that some minimal (maximal) times of the time points increase (decrease). Therefore, we have to check once again whether already-analyzed constraints are still consistent and not-redundant, or whether they can be now eliminated (line 18).

We can demonstrate the PruneDisjunctions function on the disjunctions *DI* below the Figure 44. The original disjunctions *DI* are:

1. $Dist\big(Start(T1parallel), Start(T2serial)\big) = [0,0] \vee$
   $Dist\big(Start(T1parallel), Start(T3alter)\big) = [0,0] \vee$
   $Dist\big(Start(T1parallel), Start(A5)\big) = [0,0]$
2. $Dist\big(End(T1parallel), End(T2serial)\big) = [0,0] \vee$
   $Dist\big(End(T1parallel), End(T3alter)\big) = [0,0] \vee$
   $Dist\big(End(T1parallel), End(A5)\big) = [0,0]$
3. $Dist\big(End(A1pinned), Start(A5)\big) = [0,\infty] \vee$
   $Dist\big(End(A5), Start(A1pinned)\big) = [0,\infty]$

After the pruning, we get:

2. $Dist\big(End(T1parallel), End(T2serial)\big) = [0,0] \vee$
   $Dist\big(End(T1parallel), End(T3alter)\big) = [0,0] \vee$
   $Dist\big(End(T1parallel), End(A5)\big) = [0,0]$

The disjunctions 1 and 3 are eliminated at all since the second and third constraints in the disjunction 1 and the second constraint in the disjunction 3 are not consistent with the STP graph in the Figure 45.

Proofs of Completeness and Soundness

The soundness of the PruneDisjunctions function is evident from the description above. The complexity is computed with using results of the chapter Number of iterations where it is proved that the maximal number of iterations of the used double for-loop (lines 4 and 5) is $(m+2) * n * (n-1)$ where *m* is the total number of resources in the input infeasible schedule and *n* is the total number of scheduled activities there. The complexity of adding the constraint *C'* to the STP graph (line 16) is $O(n^2)$ due to the IFPC algorithm. Further, the maximal number of constraints added to the graph equals the maximal number of disjunctions *DI* since each operation of adding eliminates one disjunction. Then, we get:

$$O\big(mn^2 * (mn^2 + n^2)\big) = O(m^2n^4)$$

The first $O(mn^2)$ stands for the maximal number of iterations of repeat-until loop; the second $O(mn^2)$ is for the maximal number of iterations of the double for-loop.

8.9.2 Sorting of Disjunctions

One of the popular heuristics for the DTP is called *Minimum Remaining Values* (MRV). The heuristic says that we should always pick a disjunction *D*, among the not-resolved-yet disjunctions *DI*, which has the fewest constraints, i.e. is the most constrained. As a result, if we continue with the solving DTP by adding one of the *D*'s constraints (consistent with the current STP graph *G*) to the graph *G* and by picking another disjunction from *DI*, we should sooner find out whether a feasible solution for that particular partial combination exists, or not ("first fail").

The Repair-DTP algorithm integrates this heuristic optimization partly. The algorithm arranges all disjunctions *DI* according to their numbers of constraints in the ascending order, but just once – before the first call of the SolveDTP function.

8.9.3 Sorting of Constraints in Disjunctions

If we take a look at the pseudo code of the SolveDTP function (see Solving of DTP), we can notice that constraints of the disjunction *D* are explored there sequentially (line 2). It follows that if the constraints are in a bad order, it may happen that the most similar feasible schedule will be the one which satisfies the last constraint of *D*. However, we want to find the most similar constraint as soon as possible since we want to provide the user a choice to stop the algorithm at any time and return the best schedule found till that time (see Repairing of Inconsistent Schedule). Therefore, the heuristic optimization has been introduced to the Repair-DTP algorithm.

The optimization sorts constraints in all disjunctions *DI* in a way that the first constraint in a particular disjunction reflects (= is similar to) the situation of the connected nodes in the input schedule the best and vice versa. Note that this optimization does not guarantee us that the first constraints of all disjunctions *DI* create a combination which is consistent with an appropriate STP graph.

8.9.4 Forward-Checking

If we apply the Pruning of Disjunctions optimization, the number of disjunctions in *DI* will be likely reduced a little; though the number may be still very high. Therefore, the Repair-DTP algorithm integrates a look-ahead approach called forward-checking [14] which prunes constraints in the not-yet-resolved disjunctions *DI* which are not consistent with the current partial combination of constraints of resolved disjunctions *DI*. As a result of this pruning, the number of iterations in the SolveDTP function and recursive calls of this function likely radically decreases.

The forward checking can be implemented pretty much the same way as the PruneDisjunctions function. However, since the implementation of the SolveDTP function is complex enough, we decided to implement just the lightweight version of the PruneDisjunctions function.

FORWARDCHECKING
Input: STP graph *G*, not-yet-resolved disjunctions *DI*
Output: pruned *DI*

```
1   FOR EACH D ∈ DI
2      FOR EACH C ∈ Constraints(D)
3         IF IsConsistent(C, G) ≠ INCONSISTENT
4            remove C from D
5         END IF
6      END FOR
12
13     IF |Constraints(D)| = 0
14        FAIL
19     END IF
20  END FOR
22  return DI
```

Figure 53: The ForwardChecking function of the Repair-DTP algorithm

Note that all constraints in all not-yet-resolved disjunctions *DI* are processed exactly once; hence the complexity of the ForwardChecking function is much better than the complexity of the PruneDisjunctions function. On the other hand, since the function does not distinguish whether the pruned disjunction *D* has one, or whether D has more constraints left, the SolveDTP function is then invoked even with disjunctions which are unambiguously resolvable, i.e. which have just one constraint. That has a bad impact on the overall processing time.

9. Experiments

In the previous chapters we have familiarized with the Repair-DTP algorithm. Now we will run the algorithm on a simple example to illustrate how the real output of the algorithm looks like and then, we will focus on testing the validity of three hypotheses concerning the algorithm's performance.

9.1 Demonstration

This section will demonstrate on small data set what we can expect from the algorithm. In Figure 54 there is an infeasible schedule of 20 nodes (10 activities) with no pins, three violated precedencies, two violated ES synchronizations, and one conflict of activities on a resource.

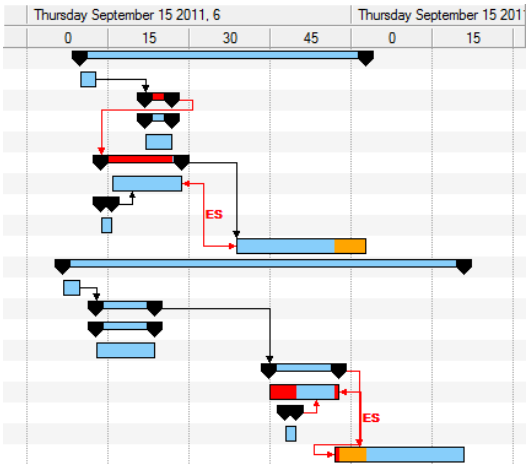


Figure 54: Input infeasible schedule without pins.

When we start the Repair-DTP algorithm, the algorithm finds a solution with rating 30 that the algorithm considers the optimal solution (see Figure 55).

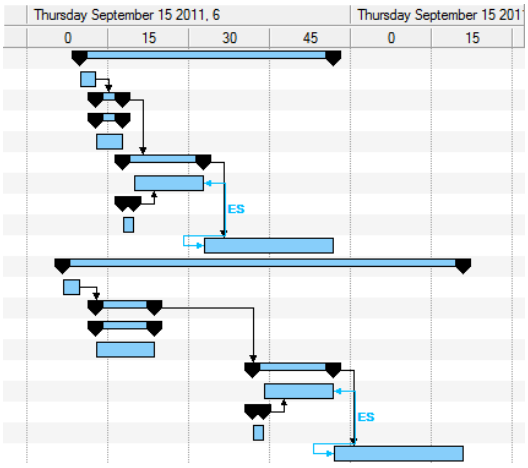


Figure 55: Resulting feasible schedule with rating 30.

We can see that all violations have been corrected successfully and we have got a very similar schedule to the original one.

9.2 Tests

This chapter presents the results of tests which are divided into three groups according to a hypothesis which the tests support. The hypotheses are:

1. The run time of the algorithm and the size of the RAM memory consumed by the algorithm grow faster than linear.
2. The run time of the algorithm is not directly dependent on the number of violated constraints in the input schedule.
3. The overall run time of the algorithm decreases when the number of pins increases.

Before we start discussing the first hypothesis, we should say that the Repair-DTP algorithm was tested on a machine with Intel Core i3-370M (2.40 GHz, 3 MB Cache), 3 GB RAM, Windows 7 Professional 64-bit, MAK€ 1.0.42.0142.

9.2.1 Hypothesis 1

The first hypothesis says that the functions of the run time of the Repair-DTP algorithm and the size of the RAM memory consumed by the algorithm grow faster than linear.

We tested the validity of this hypothesis on schedules which consisted of instances of one mutual workflow; in particular it was the workflow for manufacturing of a piston (see Figure 2). Moreover, each schedule contained only one or two violated constraints caused by a selected pinned activity (the schedule contained no other pinned activity). It followed that many nodes of the schedule had to be adjusted since the position of the pinned activity was fixed.

We conducted eight different tests: the first test worked with the schedule with only one instance of the workflow and then we successively added eight instances to each further schedule. Thus, the last test ran with the schedule of 57 instances of the workflow.

In the following chart (see Figure 56), we can see how the number of instances of the workflow influenced the number of the selected nodes in eight tested schedules and the number of disjunctions *DI* there. It is evident that the former grows linear and as it turned out, the function of the number of the disjunctions *DI* grows faster.

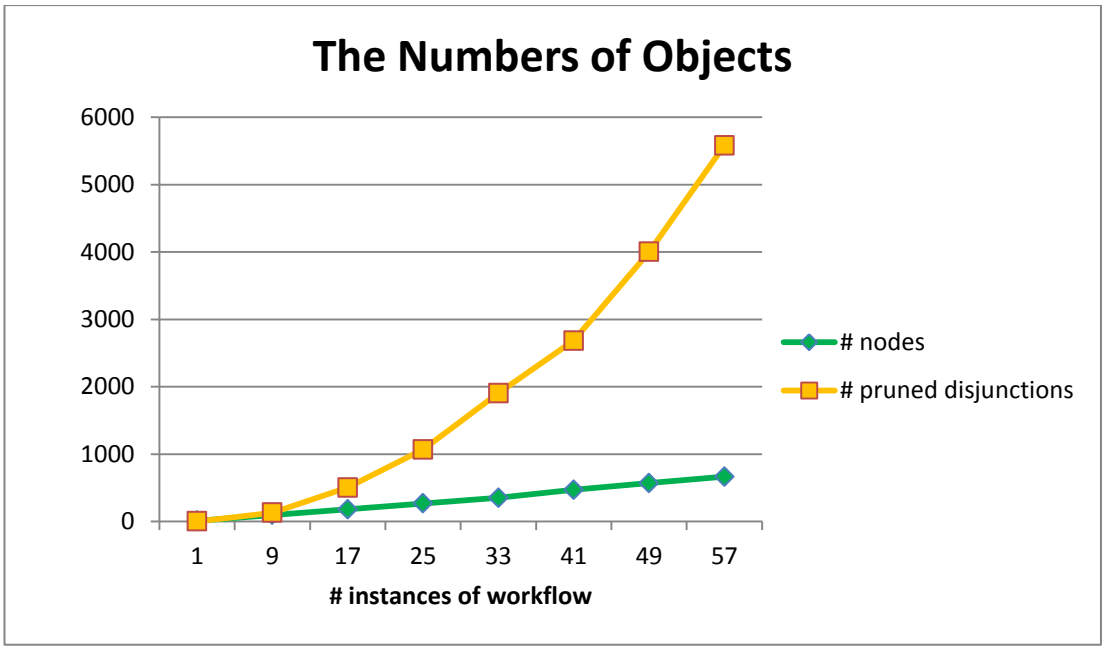


Figure 56: Functions of the number of nodes and of the number of disjunctions (pruned by the PruneDisjunctions function) depending on the number of instances of the piston's workflow in a schedule.

In the Figure 57, we can notice that the conversion of the input infeasible schedule to an appropriate DTP problem and the update of the minimal and maximal times are not a bottleneck of the algorithm (the blue function), but the problem is the run time of the SolveDTP function. When the algorithm starts solving the DTP, the exponential behavior of the function of the number of disjunctions *DI* influences the function of the overall run time of the algorithm.

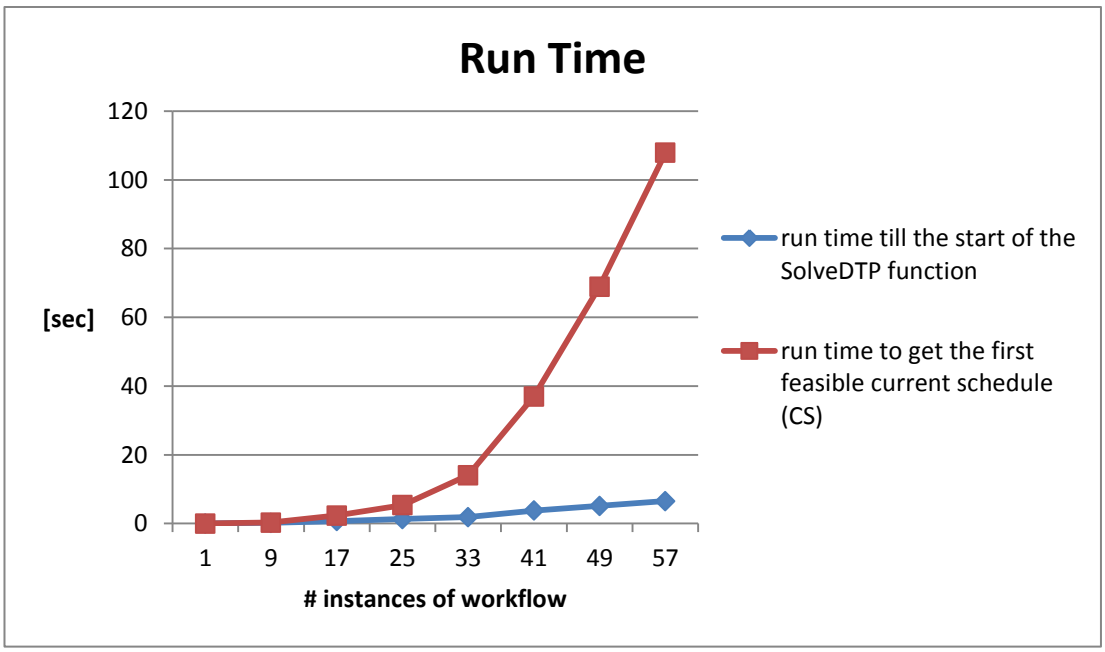


Figure 57: Functions of the run times of the Repair-DTP algorithm.

So, we have showed that the first part of the hypothesis is true. The second part of the hypothesis which discusses the consumption of the machine memory is supported by the results depicted in the Figure 58. It came out that the number of constraints in the STP graph (see Complexity) has a bad impact on the consumption of the RAM memory and the function of the consumption grows according to that.

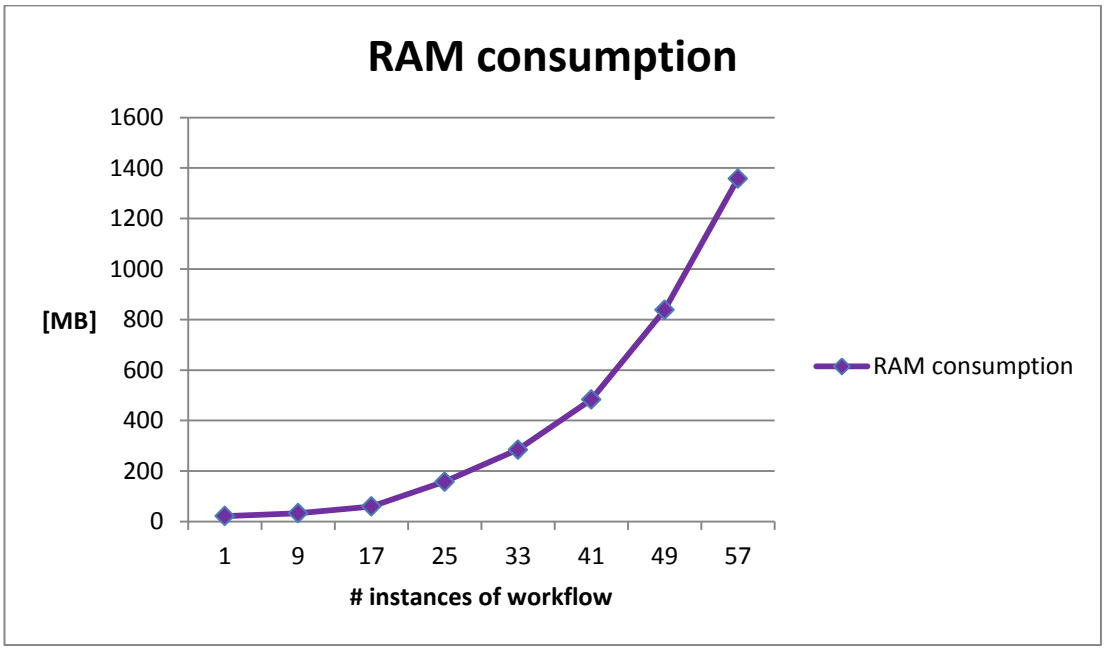


Figure 58: The RAM consumption of the Repair-DTP algorithm.

9.2.2 Hypothesis 2

The second hypothesis says that the run time of the Repair-DTP algorithm is not directly influenced by the number of violated constraints in the input infeasible schedule.

To verify the validity of this hypothesis, we used the schedule from the previous tests with 41 instances of workflows, no violated constraint and no pin.

We conducted eight different tests: the first test worked with the schedule populated by one violated constraint and then we successively added eight other violations to each further schedule. Thus, the last test ran with the schedule which contained 57 violated constraints.

In the Figure 59, we can see that the first seven tests took approximately 20 second each. However, the last one ran almost twice longer. Since all tested schedules were the same except violations, all tests explored the same combinations of constraints of disjunctions *DI*; thus they spent the same amount of time by this exploration. The differences in their run times were hence introduced by the finding of the feasible current schedules (CS). The last test spent by repairing the violated constraints in the FindCurrent function much more time than the other tests. Note that the violated constraint in the input schedule and violated constraints in the FindCurrent function

are not the same and the number of the former says nothing about the number of latter.

Figure 59: Dependence between the number of violated constraints and the run time of the Repair-DTP algorithm.

We verified the validity of the hypothesis 2. Moreover, we found out that it was much more important to know how the constraints in the input schedule were violated than how many of violations were there.

9.2.3 Hypothesis 3

The third hypothesis says that the overall run time of the Repair-DTP algorithm decreases when the number of selected pinned activities increases.

We chose a schedule which consisted of four instances of the piston's workflow to verify the validity of this hypothesis. Since we wanted to measure the overall run time of the algorithm, not just the run time till the first found solution, we took such a small schedule. Further, the schedule contained 10 violated constraints.

We conducted seven different tests: the first test worked with the schedule populated by three pins and then we successively added one other pin to each further schedule. Thus, the last test ran with the schedule which contained nine pins.

The results of the tests are depicted in the following two charts (see Figure 60 and Figure 61) and we can read there that the overall run time of the algorithm decreases exponentially when the number of pins increases linear. So, the hypothesis 3 is valid. If we combine both charts, we find out that the overall run time depends more precisely on the number of disjunctions since the schedules with seven and eight pins have the same number of disjunctions and their processing times are very similar.

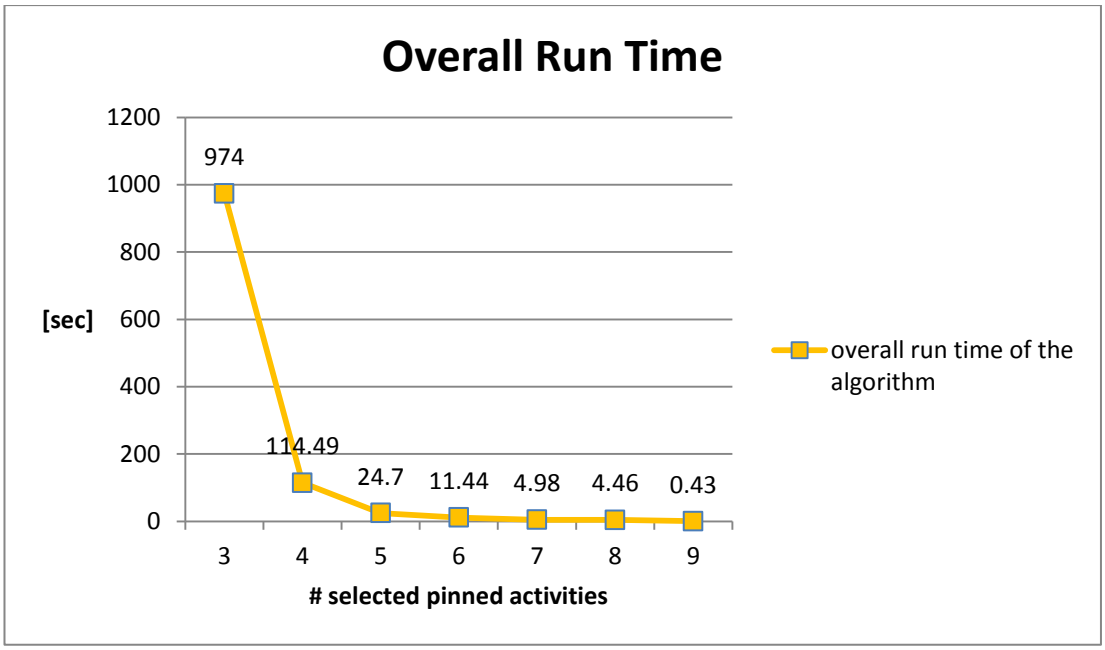


Figure 60: The overall run time of the Repair-DTP algorithm with respect to the number of pins.

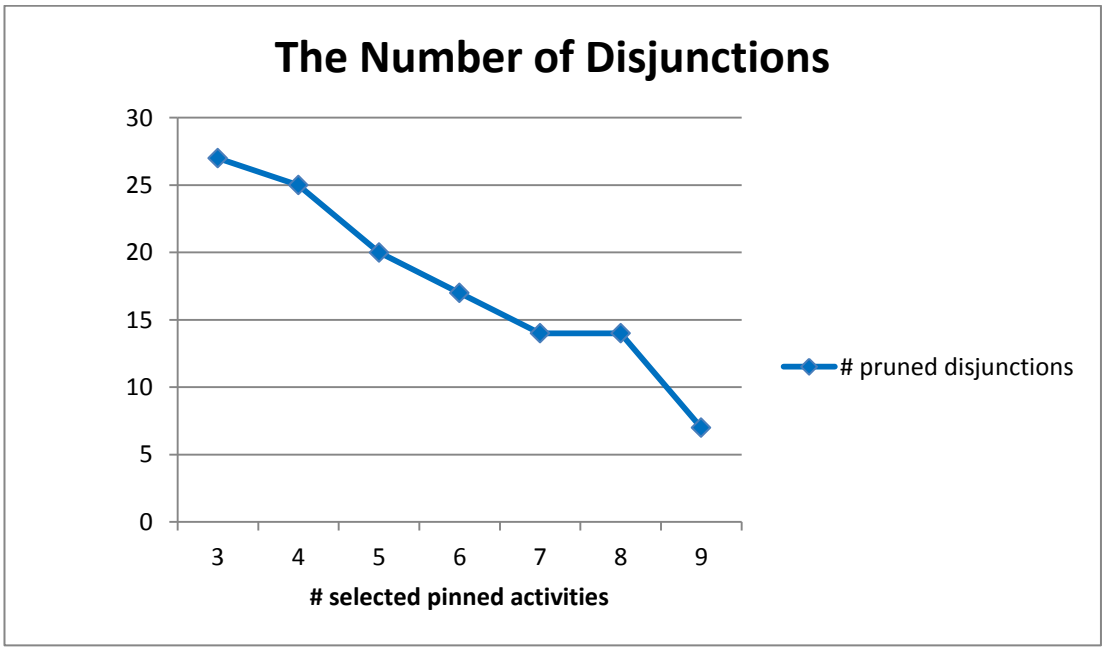


Figure 61: The tendency of the number of disjunctions (pruned by the PruneDisjunctions function) in schedules with the increasing number of pins.

In the end we should say that we successfully verified all three hypotheses. Moreover, we found even more characteristics of the behavior of the Repair-DTP algorithm which influence the efficiency of the algorithm.

Conclusion

This thesis had two goals to achieve. The first one was to design and implement an application which would visualize schedules enriched with data about associated workflows and which would enable the user to edit these schedules. The Gantt Viewer application meets both these requirements. Moreover, the operations related to the modification of schedules are mostly available through the "drag & drop" interface and the user interface is intuitive.

The Gantt Viewer application was developed as a part of the FlowOpt software project, which was successfully defended in June 2011. In the same month the whole application was already presented to potential users on ICAPS conference in Freiburg (see http://icaps11.icaps-conference.org/demos) and the feedbacks were positive. Moreover, an article [3] about the application was accepted for conference TAAI which took place in November 2011 in Taiwan and the application will probably be registered to the competition ICKEPS which is planned for June 2012 to Sao Paolo (see http://icaps12.icaps-conference.org).

The second goal of the thesis was to propose and to test in practice an algorithm which would automatically repair the violations of the scheduling constraints introduced by the user. The algorithm which solves this type of problems and was presented in this thesis is called Repair-DTP. It manages to reschedule the nodes in an infeasible schedule in a way that the obtained flawless schedule is characteristic with a high similarity to the original one.

It has been proved that the algorithm is sound, but the proof of the completeness is not complete. As it turned out both temporal and spatial complexities of the algorithm are quite high. The former grows exponentially with the number of disjunctions and the latter grows also faster than linear when the number of time points increases. For example a schedule with more than 6,000 disjunctions could not been repaired on the test machine.

Future works

The performance of the algorithm is one of the topics which emerged during the work on this thesis and which can be considered for future work. The list of topics follows:

- Proof of the lemma 2 which is missing now. The lemma is part of the proof of the completeness of the Repair-DTP algorithm.
- Improvement of pruning of disjunctions – There are three pruning rules in [13] and in the PruneDisjunctions function only two of them are utilized. The missing one is: "If a disjunction D(i) is subsumed by another disjunction D(j) then D(i) can be eliminated from D." [13].

- Usage of the backjumping – The SolveDTP function incorporates the forward-checking improvement. The next technique which would reduce the number of recursions is the backjumping [14] that represents something like a counterpart or complement to the forward-checking.
- Checking of ratings of the partial combinations in the SolveDTP function. If the rating of a particular partial combination is worse than the rating of the best solution found till that time, it makes no sense to complete the combination. Its rating would be only worse and worse (//TODO the proof of sums).
- Implementation of other Levels of Repair – The Repair-DTP algorithm represents the first and the easiest of the introduced levels. The other levels take alternative reservations and alternative branches into account. If they were implemented, banning of branches and resources would make better sense. Moreover, the user could choose which level of repairs he would like to use.

Biography

[1] ABUMAIZAR, R. J. SVESTKA, J. A. *Rescheduling job shops under disruptions.* International Journal of Production Research, Volume 35(7), pp. 2065-2082(18). 1997.

[2] BARTÁK, Roman; ČEPEK, Ondřej. *Nested Temporal Networks with Alternatives.* Hans W. Guesgen, Gerard Ligozat, Jochen Renz, Rita V. Rodriguez (Eds.): Papers from the 2007 AAAI Workshop on Spatial and Temporal Reasoning, Technical Report WS-07-12, AAAI Press, pp. 1-8 (ISBN: 978-1-57735-339-3). 2007.
Available on <http://ktiml.ms.mff.cuni.cz/~bartak/downloads/AAAI2007ws.pdf>.

[3] BARTÁK, Roman; JAŠKA, Milan; NOVÁK, Ladislav; ROVENSKÝ, Vladimír; SKALICKÝ, Tomáš; CULLY, Martin; SHEAHAN, Con; THANH-TUNG, Dang. *Workflow Optimization with FlowOpt: On Modelling, Optimizing, Visualizing, and Analysing Production Workflows.* In Proceedings of Conference on Technologies and Applications of Artificial Intelligence (TAAI), pp. 167-172, IEEE Press. 2011.

[4] BARTÁK, Roman. MÜLLER, Tomáš. RUDOVÁ, Hana. *Minimal Perturbation Problem – A Formal View.* Neural Network World, Volume 13(5), pp. 501-511. 2003.
Available on <http://www.unitime.org/papers/softcomputing03.pdf>.

[5] BARTÁK, Roman. SKALICKÝ, Tomáš. *A local approach to automated correction of violated precedence and resource constraints in manually altered schedules*. MISTA. 2009.
Available on <http://clp.mff.cuni.cz/downloads/MISTA2009paper.pdf>.

[6] DECHTER, Rina. MEIRI, Itay. PEARL, Judea. *Temporal constraint networks*. Artificial Intelligence, 49(1-3), 61–95. 1991.

[7] EL-KHOLY, Amin. RICHARDS, Barry. *Temporal and Resource Reasoning in Planning: the parcPLAN approach.* In Proc. of the 12th European Conference on Artificial Intelligence (ECAI-96). 1996.

[8] ODDI, Angelo. POLICELLA, Nicola. CESTA, Amedeo. SMITH, Stephen F. *Boosting the Performance of Iterative Flattening Search.* R. Basili and M.T. Pazienza (Eds.): AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, LNCS 4733, pp. 447–458, Springer Verlag Berlin Heidelberg. 2007.

[9] PLANKEN, Léon R. *New Algorithms for the Simple Temporal Problem*. Delft, the Netherlands, 75 p. Master's thesis. Delft University of Technology. 2008.

[10] PLANKEN, Léon R. *Temporal Reasoning Problems and Algorithms for Solving Them*. 2007.

Available on <http://www.st.ewi.tudelft.nl/~planken/Papers/literature-survey.pdf>.

[11] ROVENSKÝ, Vladimír. *Workflow Modelling*. Prague. 85 p. Master's thesis. Charles University in Prague. 2011.

[12] SKALICKÝ, Tomáš. *Interactive Gantt Charts*. Prague. 42 p. Bachelor's thesis. Charles University in Prague. 2008.

[13] STERGIOU, Kostas. KOUBARAKIS, Manolis. *Backtracking Algorithms for Disjunctions of Temporal Constraints.* AAAI-98, Madison, WI. 1998.

[14] TSAMARDINOS, Ioannis. POLLACK, Martha E. *Efficient solution techniques for disjunctive temporal reasoning problems*. Artificial Intelligence, 151, 43–89, Elsevier B.V. 2003.

[15] VERMIROVSKÝ, Kamil. RUDOVÁ, Hana. *Limited Assignment Number Search Algorithm.* In Maria Bielikova (ed.): SOFSEM 2002 Student Research Forum, 53-58. 2002.

[16] VIEIRA, Guilherme E. HERRMANN, Jeffrey W. LIN, Edward. *Rescheduling manufacturing systems: a framework of strategies, policies, and methods.* Journal of Scheduling 6: 39-62. Kluwer Academic Publishers. Printed in the Netherlands. 2003.
Available on
<http://cepac.cheme.cmu.edu/pasi2011/library/henning/ReschedulingFramework-VieiraHerrmannLin.pdf>.

Table of Figures

List of Abbreviations

(Nested) TNA	Nested Temporal Networks with Alternatives
DTP	Disjunctive Temporal Problem
EE (synchronization)	End-to-End synchronization
ES (synchronization)	End-to-Start synchronization
GUI	Graphic User Interface
IFPC (algorithm)	Incremental Full Path Consistency (algorithm)
IFS	Iterative Flattening Search
LAN (algorithm)	Limited Assignment Number (algorithm)
MRV	Minimum Remaining Values
MUTEX	Mutual Exclusion
PredRep (algorithm)	Precedence Repair (algorithm)
RSR	Right Shift Rescheduling
SE (synchronization)	Start-to-End synchronization
SS (synchronization)	Start-to-Start synchronization
STP	Simple Temporal Problem