Free University of Bozen-Bolzano

Master Thesis in Computer Science

Processes in Construction: Modeling and Consistency Checking

Matthias Perktold

supervised by
Prof. Werner Nutt
Dr. Elisa Marengo

March 21, 2017

Acknowledgments

I would like to thank the people who supported me in writing this thesis:

- Werner Nutt and Elisa Marengo, my supervisors, for the time invested in guiding me through the process.
- ASA, the company I am working in, for providing me great flexibility throughout the master study program.
- My family, for standing behind me, and my father also for our discussions on the topic.
- Daniela Figl, my girlfriend, for her patience and emotional support.

Matthias Perktold

Abstract

Project management in construction is difficult because of several aspects to be considered, such as: *i)* different companies with different business objectives have to interact and synchronize their activities, *ii)* unexpected events need to be addressed, *iii)* customer requirements often change while the construction process is running, *iv)* resources are shared among different companies and their usage need to be maximized. To support project management, a first step is the definition of a process model, which captures the tasks to be performed, the needed resources, and the collaboration among the different companies, in terms of dependencies among the tasks. To this aim, a group of civil engineers introduced the PRECISE methodology for process management and applied it successfully in real cases. It defines a graphical modeling language tailored to the construction domain. However, there is only limited and ad-hoc IT support.

The focus of this thesis is to provide (automatic) support for the PRECISE process modeling. More precisely, we focus on two main aspects: *i)* provide support for the graphical definition of a process model; *ii)* provide automatic support to check properties of interest on a model. Specifically, we address the problem of consistency of a model (i.e. whether there exists a process execution that satisfies all requirements expressed in a process model). To this aim, we developed a web application and algorithms for the consistency checking. We compare these algorithms with more standard techniques, such as consistency checking of a set of Linear Temporal Logic (LTL) formulas using model checking techniques.

Contents

1 Introduction **2**

2 Related Work **4**
2.1 Project Management in Construction 4
2.2 Process Modeling 7
2.3 Software Tools 7

3 The PRECISE Methodology **9**
3.1 Overview 9
3.2 PRECISE Framework 9
3.3 Process Modeling 10

4 Process Modeling Proposal **13**
4.1 Configuration Part 13
4.2 Flow Part 15
4.3 Formal Semantics 20

5 Checking Consistency **27**
5.1 Consistency of Basic Models 27
5.2 Consistency of Exclusive Models 30
5.3 Consistency of General Models 35
5.4 Checking Orientability 38

6 On Scheduling **55**
6.1 Checking Conformance 55
6.2 Generating Schedules and CPM 57
6.3 Scheduling Complexity 59

7 Application **61**
7.1 Requirements 61
7.2 Architecture and Design 64
7.3 Implementation 74
7.4 Validation 74

8 Conclusion and Future Work **77**
8.1 Summary 77
8.2 Future Work 77

Chapter 1

Introduction

The construction industry is one of the main fields of economy worldwide as well as in the province of Bolzano. This sector mostly consists of small and medium sized enterprises (SMEs), both in Bolzano and Italy in general. Therefore, construction projects usually involve several SMEs working together. Each of them is specialized in one sub-domain of construction, and therefore has a different set of skills and goals. To achieve the desired outcome, the involved companies need to exchange their knowledge and agree on how to collaborate. Thus, collaboration among the companies is challenging and requires careful process management, and in particular process modeling.

Several well-known general purpose methods exist for business process management and process modeling, such as BPMN (Object Management Group, Inc, 2011), the Critical Path Method (CPM) (Kelley Jr and Walker, 1959), PERT, and Gantt charts as well as tools such as Microsoft Project and Oracle Primavera. However, using these general purpose techniques in construction has several disadvantages: First, they only provide generic abstractions for modeling the processes which are common to all domains, so they are not able to capture the details of the construction domain. Second, they are not designed for collaboratively modeling processes across different parties. Third, the tools do not provide a clear separation between an abstract process model and a concrete execution plan.

Consequently, schedules are not reliable, because they lack detailed information, and do not consider well any potentially conflicting goals among involved parties. During the project, this makes it hard to discover delays and cost overruns before it is too late to compensate. Also, implementing such compensations is hard due to the lack of an abstract process model.

We can summarize that effective process management in construction needs to cope with several difficulties that do not exist in for projects of other domains. One is the high number of details to be captured, such as the tasks to be performed, the locations where they are to be performed, resources needed, and dependencies among tasks. Another one is the involvement of multiple different parties. Moreover, projects are non-repetitive, i.e. each project is unique and requires a high amount of creativity. Additionally, projects are unpredictable due to weather conditions, damaged resources, and changing requirements. Finally, process management should be made as easy as possible, since SMEs cannot invest their limited resources into expensive training.

To tackle these problems, a group of civil engineers introduced a new domain-specific methodology called PRECISE (Process REliability in ConstructIon for SmEs) by re-engineering completed construction projects (Dallasega et al., 2013). PRECISE divides the project into three phases: process modeling, scheduling, and monitoring. In the modeling phase, the involved companies meet in a collaborative modeling workshop. The goal of the workshop is to agree on a process model, which is represented using a new declarative and domain-specific graphical language that captures all the details such as tasks, locations and dependencies. Such a detailed process model enables detailed scheduling and detailed monitoring, and allows to quickly react to deviations. The methodology has been successfully applied in real projects.

In this thesis, we focus on the process modeling part of PRECISE. While graphical language introduced by PRECISE is able to model the domain and its details, it is ambiguous and lacks formal semantics. Only persons who have a background in construction and participated in the workshops in which the process model was designed are able to understand the exact intended meaning. In particular, as long as there is this ambiguity, it is not possible to develop automated

tools that support process modeling, scheduling, or monitoring.

The contribution of this thesis is to provide a basis for IT-support for PRECISE process modeling, and is two-fold. The first part of the thesis if of theoretical nature, where we define a formal semantics for the modeling language to eliminate all ambiguities. This gives us a sound reasoning base, which we use to show how to check consistency of process models. The second part of the thesis concerns the development of a web application that supports process modeling using the proposed modeling language. In particular, the application allows to draw and persist process models that can be defined both graphically and textually. Additionally, it supports consistency checking based on the formalization and the algorithm introduced in the theoretical chapters.

The thesis is structured in the following way. Chapter 2 summarizes existing work on process management in construction and process modeling in general. Chapter 3 discusses the original PRECISE methodology in more detail. Our contribution starts in Chapter 4, where we present our proposal for process modeling as an extension of the PRECISE process modeling language with formal semantics. In Chapter 5, we propose an algorithm that checks consistency of a process model by means of a newly introduced graph representation, and present experiments on an implementation. Other possible uses of that graph representation, which are more related to scheduling, are discussed in Chapter 6. In Chapter 7, we provide an overview of the web application that we developed to support process modeling and consistency checking. Finally, we conclude the thesis in Chapter 8.

Chapter 2

Related Work

2.1 Project Management in Construction

There exist two kinds of methods for project management in construction: *activity-based* methods and *location-based* methods (Kenley and Seppänen, 2009, 2006).

Traditional approaches The traditional methods are activity-based, i.e. they focus on a network of activities connected by logical dependencies. Each activity has a duration, which, depending on the particular approach, is specified either as a fixed value or by random variables with a given distribution. Also, activities are free to move in time up to the limit prescribed by successor and predecessor relationships in the network.

One popular activity-based method is the Critical Path Method (CPM) (Kelley Jr and Walker, 1959). An activity network can be constructed by representing activities as nodes and precedences as arcs between such nodes, and a duration estimate is assigned to each activity. Starting from the project beginning at time zero, a *forward pass* computes the earliest start and end times for each activity as well as the earliest end of the project possible. Then, a *backward pass* starts at the calculated end time and goes back towards the start of the project, computing the latest start and end times for each activity. The result is a flexible schedule indicating time ranges for when each activity must be started and completed.

Finally, one can compute the *float* of an activity, i.e. the amount of time by which it can be delayed without delaying the whole project. Activities with zero float are called *critical*. A path from the start to the end of the project that only visits critical activities is called the *critical path*. The project manager should pay special attention to critical activities, and add more resources to accelerate them if necessary.

Another popular activity-based method is the Program Evaluation and Review Technique (PERT). It is very similar to CPM, albeit serving another purpose. It also involves the construction of a network, but rather than searching for the critical path, the goal is to obtain the probability of completing the project by a fixed deadline. Since this is not possible with only one fixed and deterministic duration estimate for each activity, one must provide three different estimates: a most likely duration, a best-case duration, and a worst-case duration. Thus, while CPM is about project planning, PERT is about project control.

In addition to activity-based analytic methods such as CPM and PERT, another very widespread tool is the Gantt chart. A Gantt chart graphically represents activities as horizontal bars on a timescale. Additionally, a Gantt chart may also include milestones. The strength of Gantt charts lies in their simplicity and easy-to-read representation. However, in contrast to CPM and PERT, Gantt charts do not consider a logical network of activities. Thus, they provide a static view of a schedule, without any explanation of why a particular sequence of activities was chosen and whether some alternative sequence could be used as well or not. Therefore, they are not well suited as a main tool for the planning process, but more for communicating the resulting schedules. Indeed, Gantt charts are the most popular scheduling communication format in practice (Kenley and Seppänen, 2006).

Activity-based techniques such as CPM and PERT are general purpose, and they have proven useful for managing projects in various domains, including construction. Yet, being general purpose, they are not able to represent the details of the construction domain in a structured way.

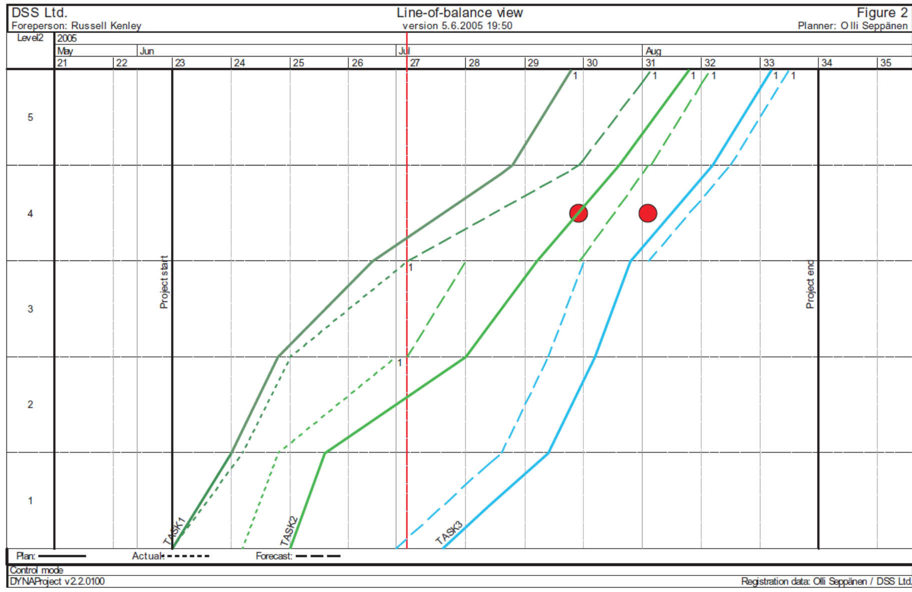


Figure 2.1: A flowline chart showing planned, actual, and forecast schedules (Seppanen et al., 2005). Each solid line of different color represents the planned movement of a crew while performing a task. Dotted and dashed lines represent the actual and forecast performance of the respective crew-task combination of the same color.

In particular, they do not take into account that in construction, activities are executed in multiple locations on the construction site. One option to deal with this in an activity-based setting is to agree that each activity represents work to be executed in all locations. This can lead to sub-optimal schedules, since an activity cannot start before its predecessor activities finished completely, i.e. in all locations. Often, it is sufficient to only impose such a precedence constraint within the same locations, so when an activity finishes in one location, the successor activity can start, while at the same time the previous activity progresses with the next location. Further, the progress cannot be monitored in detail, so delays are discovered late. Another option to deal with locations using an activity-based method is to define activities as a work package to do in a particular location, i.e. for each work package, there is a separate activity for each location where the work is to be performed. One problem with this approach is that the resulting networks become huge and therefore hardly manageable. Also, activity-based methods do not optimize continuity of work flow, but only focus on earliest completion time. The consequence is that the work of a crew is randomly scattered across time and locations, resulting in inefficient resources usage.

Innovative approaches There are several approaches for improving the management of construction projects: location-based methods, Building Information Modeling, and lean construction.

Location-based methods address the problems of activity-based methods by explicitly taking into account the special role which locations play in construction projects. In particular, they not only focus on earliest completion date, but also consider continuous work flow. They are based on tasks that repeat in multiple locations, and track the continuity of crews moving through locations as they perform a task

One representative location-based scheduling method is the *flowline* method. See Fig. 2.1 for an example. A graphical schedule indicates how crews move through the building while performing a task by plotting these movements as lines in a grid where locations are listed on the vertical axis and time corresponds to the horizontal axis. The aim is to achieve production efficiency by ensuring continuous work flow and well aligned production rates, and these factors are well illustrated in the flowline diagram: The work is continuous if lines are not interrupted, and production rates are well aligned if all lines have equal or similar slope.

Kenley and Seppänen (2006) integrated the flowline method into the Location-Based Management System (LBMS). LBMS is an integration of multiple location-based techniques into one coherent system, supporting various production stages. A Location Breakdown Structure (LBS) defines the locations of a project in terms of a hierarchy, breaking down larger areas into smaller

sub-areas. This is similar to the Work Breakdown Structures (WBS) used in activity-based methods, but applied to locations instead of activities. Furthermore, quantities and resource consumption rates are assigned to locations at various levels in the LBS to form the knowledge base for all techniques involved. The flowline method covers scheduling, and a flowline schedule can be generated using *layered CPM logic*, i.e. logical dependencies at various levels within the LBS hierarchy. Moreover, if the process is adequately monitored regarding actual quantities, resources, and durations, flowline can be used to display and compare the original plan (*baseline*), the *current* plan, the actual *progress* as well as a *forecast*. Figure 2.1 shows a flowline schedule with separate lines for planned, actual, and forecast performance. The forecast is calculated by assuming that crews will continue their work with the current actual production rate (Seppanen et al., 2005). If this is not possible due to the layered CPM logic, the tasks are shifted accordingly and a warning is produced.

Another approach is the Building Information Modeling (BIM). The National Building Information Model Standard Project Committee and others define it in the following way:

> Building Information Modeling (BIM) is a digital representation of physical and functional characteristics of a facility. A BIM is a shared knowledge resource for information about a facility forming a reliable basis for decisions during its life-cycle; defined as existing from earliest conception to demolition. (National Building Information Model Standard Project Committee and others)

The *dimensions* of a BIM indicate the types of information included (Out-Law, 2012). A 3D model only includes the spatial design of components and can be used to identify design clashes. When a time dimension is added, we get a 4D model which allows to simulate the construction of the building. The fifth dimension includes cost, material quantities, and crew productivity. Thus, a 5D model enables one to automatically adjust estimates when something changes, and to perform value engineering in the design. A 6D model further contains energy consumption information for managing the sustainability of a building. This is especially useful when combined with sensors to get data in real-time. Finally, a 7D model supports facility management by considering also maintenance manuals and schedules, warranty information etc.

To gain the most benefit out of a BIM, it should cover as many dimensions as possible. Then all companies involved in the project can operate on the same model. This enables one to ensure that data is always consistent and up-to-data data.

The dominant philosophy behind recent research in project management in construction is covered by the term *lean construction*, which is the result of applying lean production theory from manufacturing to the construction sector. The main principle behind lean production is to increase productivity by eliminating waste.

In traditional approaches, activities are mainly scheduled so as to achieve fixed deadlines. This is called a *push* system because activities are released (pushed) from the master schedule to the construction site based on due dates only. Lean construction instead favors a *pull* system, where work is released (pulled) from the master schedule to the construction site based on the current state on site, in addition to due dates. This leads to a conception of project control where deviations from the plan are detected "after the fact" and only then corrective actions are initiated.

One notable approach to lean construction which is pull-based is the Last Planner System (LPS) (Ballard, 2000). It focuses on production control, which in this context is defined as "to cause events to conform to plan" (Ballard, 2000), as opposed to the traditional conception of project control based on an "after the fact" detection of variances between the plan and the work actually performed. Several key concepts contribute at achieving this goal that are best summarized as *should-can-will-did*. A *master schedule* defines what *should* be done in terms of milestones. This is further refined into a *phase schedule* (or *pull schedule*) by starting from milestones and defining which activities are required to be finished in order to meet the milestone, and further working backwards in time until the preceding milestone. To increase the reliability of such plans, they are defined in collaborative workshops involving trades who are to perform the actual work on-site. Then, a *look-ahead schedule* contains work that *can* be done in the next several (typically six) weeks at the level of more detailed assignments. To ensure that these assignments really can be done, an active process checks whether requirements for assignments are met on the construction site and removes any obstacles found.

This enables a *last planner* to choose only assignments that *can* be done and create a plan of what *will* be done for the upcoming week. Finally, the ratio of completed assignments (*did*) to those

planned by the last planner indicates the reliability of plans. The reasons for failed completion of assignments should be investigated and removed, resulting in a higher reliability of plans.

Seppänen et al. (2010, 2015) propose a combination of LPS and LBMS, where social processes such as collaborative planning are taken from LPS, while the technical tools such as the flowline method are taken from LBMS. The combined approach also combines the benefits of both its underlying systems.

While LPS and LBMS as well as their combination are promising approaches for scheduling and control, they do not consider process modeling in a satisfactory way. Similarly, the focus on BIM is more on modeling the product, i.e. the building, rather than on the processes. Still, when it comes to scheduling the work to be performed on-site, the process becomes important. Therefore, our approach is to be understood as complementary to BIM. In particular, we claim that collaborative process modeling can greatly improve the information flow among all parties involved in a construction project, and therefore result in more reliable schedules.

2.2 Process Modeling

General approaches to process modeling that are well-known include the *Business Process Modeling Notation* (BPMN) (Object Management Group, Inc, 2011) and *Petri Nets*. Both can be represented in a graphical way and focus on the control flow, i.e. they define the possible sequences in which activities can be executed. Unfortunately, they fail to capture the details that are specific to construction projects, such as locations, because they aim at supporting all domains and thus can only capture the details common to all domains, such as activities. Therefore, they do not suit our particular needs.

A different approach to a language for process modeling is that of *ConDec* (Pesic and Van der Aalst, 2006) and *DecSerFlow* (Declarative Service Flow Language) (Aalst and Pesic, 2006). ConDec was developed to support modeling and enacting dynamic business processes, while DecSerFlow is targeted towards web services. These languages are declarative, which is fundamentally different from procedural languages such as BPMN and Petri Nets. In general, a declarative language only describes *what* the desired result is, while a procedural language describes *how* to obtain it. In the context of process modeling, the desired result is a valid sequence of activities. Thus, BPMN and Petri Nets describe how to obtain a valid sequence of activities, whereas ConDec and DecSerFlow only specify constraints that a sequence of activities must satisfy to be valid.

The advantage of procedural process modeling languages is that it is easier to construct a sequence from a given model. However, because sequences need to be defined explicitly, often some sequences that are actually valid are not considered in the model, i.e. the model is over-specified. Consequently, the model needs to be updated whenever a particular possibility is identified to be missing. This is a disadvantage compared to declarative process modeling languages, which only model constraints on possible sequences. In ConDec and DecSerFlow, these constraints are defined using a graphical language which is grounded on linear temporal logic. The language can potentially be extended by defining new constraint types and the corresponding LTL formulas. This shows the great flexibility that comes with the declarative approach of ConDec and DecSerFlow.

On the other hand, ConDec and DecSerFlow also aim at supporting all domains, and therefore do not capture the specifics of construction. Thus, they are not ready to be used for our purpose. For this reason, we define a new process modeling language specifically for construction projects. Still, we took inspiration from the declarative approach of ConDec and DecSerFlow to benefit from their advantages.

2.3 Software Tools

Two of the most commonly used software tools for project and process management in construction are Microsoft Project and Oracle Primavera. They are based on traditional activity-based methods such as CPM and PERT and thus share the problems of their underlying methodologies when applied to construction projects.

Many software solutions are available for producing a BIM, including Graphisoft ArchiCAD, Autodesk BIM 360 and Revit, and Tekla Structures. There are also other software packages which do not focus on creating a BIM, but on taking advantage of existing BIMs instead. Vico Office Suite, for instance, offers a 5D solution based on an imported 3D BIM model that can be visualized

and used for scheduling, showing 4D simulations of a schedule in a building, and for cost estimating (Vico Software; Kenley and Seppänen, 2006). In particular, the Vico Office Suite not only tackles BIM, but it is also based on location-based methods for planning and control. There is also a "Dependency Network" view, that graphically depicts a network of tasks and dependencies. This is not included in the underlying LBMS, and it is one step towards process modeling. Yet, the view does not show all the details such as the level in the LBS to which a dependency applies, or the locations and their order in which a task has to be executed. Thus, it is not possible to understand the whole logic of the process by just looking at it, but that is a requirement for collaborative process modeling.

Chapter 3

The PRECISE Methodology

In this section, we present the original PRECISE methodology (Dallasega et al., 2013), where PRECISE stands for Process REliability in ConstructIon for SmEs. It emerged from the research project build4future (Matt et al., 2011), which was launched to develop better planning approaches for supporting local SMEs in the Architecture, Engineering, and Construction (AEC) sector. We will first give an overview of PRECISE in general and then discuss how it could be supported using IT. Finally, we will focus on the PRECISE approach for process modeling and identify its shortcomings.

3.1 Overview

Traditionally, a planner defines task sequences and durations without consulting the executing companies. This represents a *push* methodology and often leads to imprecise or unreliable workflows, because requirements and actual capacities of the execution companies are not known and therefore not considered. PRECISE aims at improving the reliability of processes by actively involving all participating companies in the process modeling. This collaborative approach supports mutual understanding of processes and requirements among the companies. More precisely, PRECISE distinguishes three phases that require collaboration in different ways:

1. The Early Interdisciplinary Building Design (EIBD) phase covers the design of the building. The idea is that the executing companies should contribute to the building design by evaluating and optimizing it with respect to accessibility, constructability, durability etc. Thus, product and process design are integrated at this phase. To make this possible, the participating companies should be known as soon as possible.

2. The Integral Building Execution Planning (IBEP) phase concerns the definition of a process plan. The companies involved in the project collaboratively define the process plan by agreeing on how to coordinate their work. They are supported by a neutral moderator who creates the plan in the format expected by PRECISE and extracts the relevant information from the participants.

3. The Dynamic Plan and Control Station (DPCS) is a tool for coordinating the work among participants on and off-site. Construction progress is scheduled and controlled in short time intervals, and the progress is made transparent to all participants.

While the EIBP phase is an important success factor, implementing it in the AEC sector is difficult because of contractual and "public procurement law requirements, diverging project objective, and a lack of process understanding" (Dallasega et al., 2013). Therefore, the methodology and in particular the planned IT support focus more on the last two phases.

3.2 PRECISE Framework

Dallasega et al. (2015) propose a framework consisting of three interconnected components for supporting the IBEP and DPCS phases of the PRECISE methodology. The components and their connections are shown in Fig. 3.1. The *modeling* component defines a graphical process

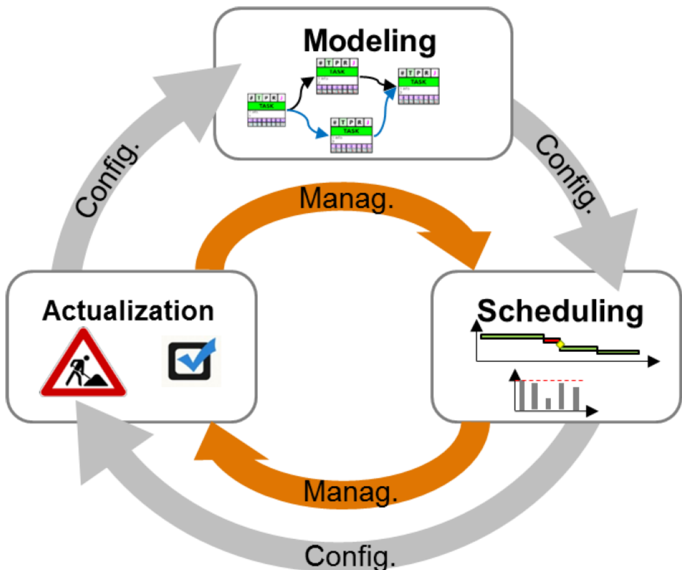


Figure 3.1: The three components of the PRECISE framework (Dallasega et al., 2015).

modeling language specifically designed for the construction domain to support the IBEP phase. The language is described in more detail in Section 3.3. The resulting process model defines tasks, locations, and dependencies among them in the project. This information is used to configure the *scheduling* component to assign tasks to crews to schedule them for specific time periods. This covers one part of the DPCS phase. The other part is covered by the *actualization* component, which allows to report the actual progress of the scheduled work. This data is then used for the next scheduling iteration, and sometimes also to improve the model, e.g. by adjusting productivity rates or introducing new tasks.

3.3 Process Modeling

One important aspect in PRECISE is a clear distinction between process modeling and scheduling. A process model defines the tasks to be performed, the locations in which the execution of the tasks takes place, and dependencies among tasks. In a schedule, each location of each task has start and end dates as well as the crew that performs the work. The process model sets the frame for scheduling: all tasks should be scheduled in all locations, and dependencies should be satisfied. A process model is particularly useful for re-scheduling to compensate deviations from the plan: every schedule that satisfies all constraints specified in a model is allowed. If a schedule is not accompanied by a model, it would not be clear whether some particular change in the schedule, e.g. switching the order in which two tasks are executed, is allowed or not.

To ensure a high reliability of process models, they are defined in a collaborative workshop by all companies involved in the project. For that purpose, a new domain-specific graphical language is used. In particular, the language integrates locations of tasks into the process model by introducing the concept of construction areas and construction phases (Dallasega et al., 2015).

Building Model A *construction area* (CA) identifies a part of a building where tasks of a particular *construction phase* can be executed. There are three construction phases:

1. *Skeleton*: erection of the basic building structure.
2. *Envelope*: installation of the facades.
3. *Interior*: erection of the internal building part.

Each phase requires a different representation of CAs. In the skeleton phase, a coarse granularity is sufficient. Here, a CA is defined by a *sector* and a *level*. A sector represents a part that can be managed independently, such as a wing of a building, or even a separate building. In the envelope phase, CAs are defined by *sector*, *orientation*, *level*, and *unit*. The orientation refers to the cardinal

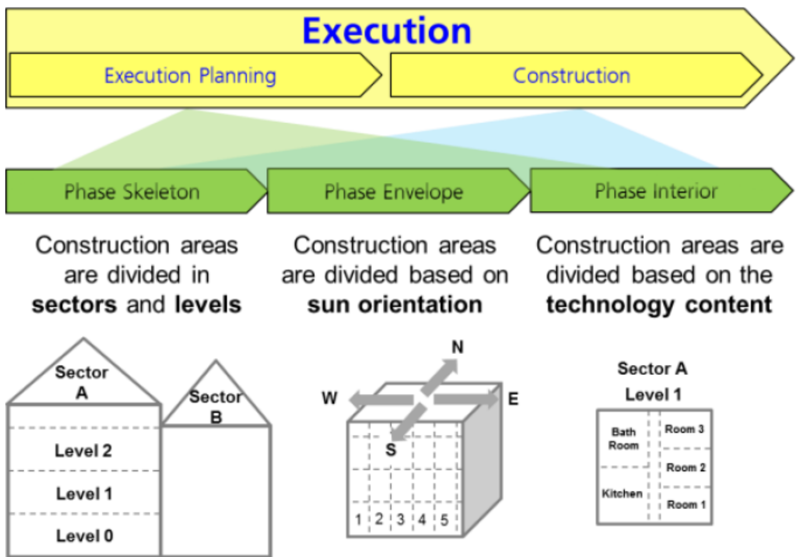


Figure 3.2: Dividing a building into construction areas (Dallasega et al., 2015).

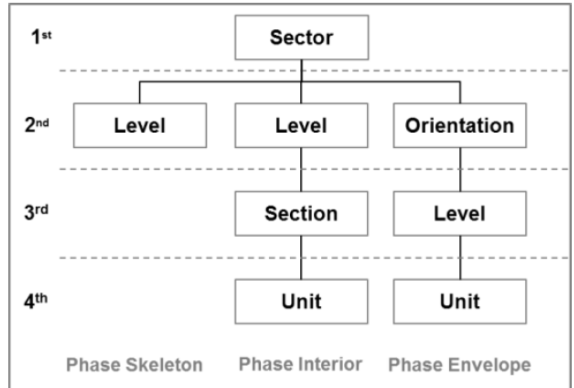


Figure 3.3: Hierarchy of construction areas for different phases (Dallasega et al., 2015).

direction of a facade, i.e. North, East, South, or West. Each level of a facade is further divided into small slices, called *units*. In the interior phase, CAs are represented as *sector*, *level*, *section*, *unit*. Here, the section identifies the technological content of a CA, i.e. whether it is a room, a kitchen, a corridor, a swimming pool etc. Units are used to distinguish different locations of the same section. Figures 3.2 and 3.3 illustrate how the different phases divide buildings into CAs.

Process Model Given a model of the building in terms of CAs, we can define a process model as a network of tasks that are connected by dependencies with information on the CAs where tasks are to be performed. Figure 3.4 shows a graphical representation of a process model for the realization of a hotel. Boxes represent tasks with relevant information. The header row of a box contains ① a unique ID, ② the number of required workers, ③ the expected time spent on this task in total, and ④ the craft responsible for executing the task. Next, there is ⑤ the name of the activity to be executed, and ⑥ a space for additional descriptions and comments. Finally, at the bottom of a box, a table represents the locations where the activity is to be performed. Rows correspond to CA attributes, and columns correspond to locations. That is, each column refers to one location by specifying values for each CA attribute. In this example, the CA attributes are ⑦ section and ⑧ level.

Shortcomings As discussed previously, a process model provides a framework in which scheduling can act. However, there are several ambiguities in the modeling language that hinder this approach.

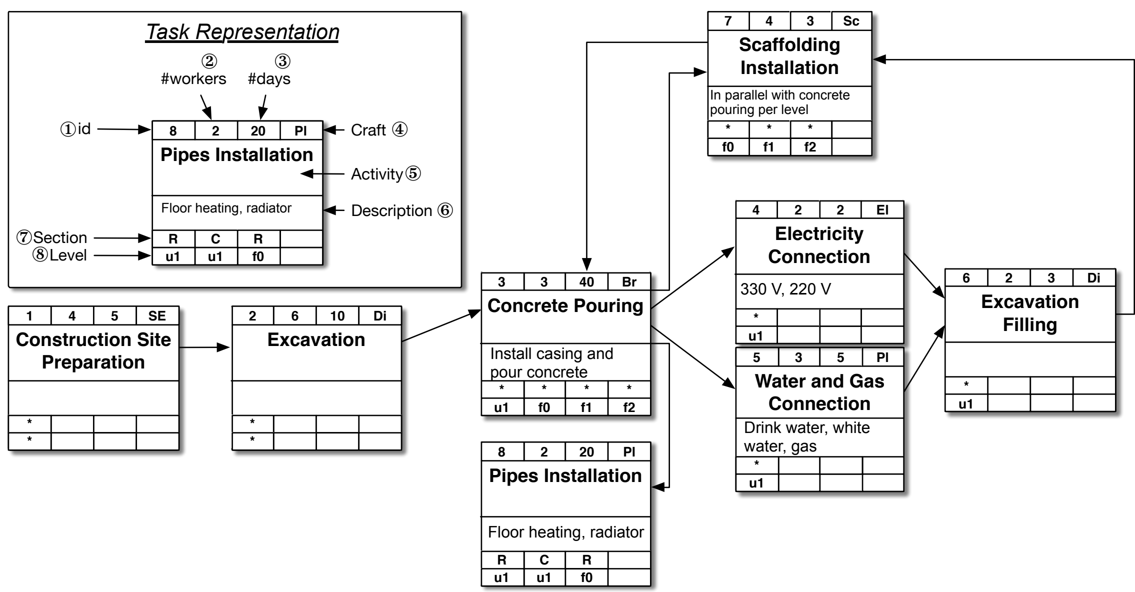


Figure 3.4: Excerpt of a real process model for the realization of a hotel based on the original notation. The wild-card $*$ represents all sections and/or levels. In the project, there are four floors (u1, f0, f1, f2). The sections are Room (R) and Corridor (C). The tasks also indicate involved crafts, which are Site Equipper (SE), Digger (Di), Brick Layer (Br), Plumber (Pl), Electrician (El), and Scaffolder (Sc).

1. Execution order of locations for a taks: Sometimes the order in which a task is executed in its locations cannot be decided freely. For example, Concrete Pouring must start in the lowest level and proceed with the next higher one until the top-most level is reached. Such ordering constraints cannot be captured in the model.

2. Scope of a dependency: For two tasks connected by a dependency, it is not clear whether one task must be performed in all locations before the other can start in any location, or whether only within each shared location, one must be completed before the other can start. For example, the intention of the dependency from Construction Site Preparation to Excavation is that the whole task must be finished, i.e. in all locations, before the next task can start. Conversely, the dependency from Scaffolding Installation to Concrete Pouring should apply at the scope of levels, i.e. the second task can start in a level as soon as the previous task is finished in all locations of that level. This ambiguity is particularly apparent for cycles, because we do not even know which task comes first.

3. Definition of dependency: Dependencies represent precedence constraints among tasks. Tasks are not atomic, but have a duration, and therefore a start and an end. It is not defined whether dependencies represent end-to-start, start-to-start, end-to-end, or start-to-end relations.

Most ambiguities can be resolved by applying construction knowledge, remembering what was discussed verbally during the workshop, or putting extra information in the description fields of tasks. But when it comes to IT-support, such workarounds are not possible. Thus, in order to enable IT support, we need to overcome these ambiguities by extending the language when needed and introducing a formal semantics. This is what we will do in the next section.

Chapter 4

Process Modeling Proposal

In Section 3.3, we presented the original approach to process modeling in PRECISE and pointed out some limitations of that approach. In this section, we propose an extension to the original approach meant to overcome those limitations.

This extension divides a process model into a configuration part and a flow part. The configuration part defines activities that *can* be executed and areas in the building where activities *can* take place, while the flow part specifies the areas in which an activity *must* be executed, as well as constraints such as temporal dependencies among CUs and among activities that *must* be satisfied. We describe the involved concepts together with a graphical representation in more detail in the following sections. Then, we provide a formal semantics for the various constraints.

4.1 Configuration Part

The configuration part contains static information of the process model. It structures the building into abstract CUs and describes activities that can be executed in these CUs. Moreover, this part contains construction phases, each with there own representation of CUs.

4.1.1 Building Structure

One important part of the process configuration is the building structure. We take the basic concepts from PRECISE, but provide more flexibility, such that for each project, one can define the structure that best fits the requirements. The building structure is defined in terms of *attributes* such as wing or level. An attribute has a name and a *range* of values. For example, the following could be used to describe a building with two wings A and B as well as four levels from the first underground floor to the second upper floor:

$$\begin{aligned} \mathit{sector} &:= \{A, B\} \\ \mathit{level} &:= \{-1, 0, 1, 2\} \end{aligned}$$

In a process configuration, there is one global set of available attributes. We use relations over these attributes to define *construction units* (CUs). A CU abstractly represents a place in the building where some work needs to be performed as a combination of attributes and attributes values. Continuing the previous example, we can describe CUs in terms of the attributes *sector* and *level* by providing the following relation Cu:

$$\begin{aligned} &Cu \subseteq \mathit{sector} \times \mathit{level} \\ &Cu = \{\langle A, 0\rangle, \langle A, 1\rangle, \langle A, 2\rangle, \langle B, -1\rangle, \langle B, 0\rangle, \langle B, 1\rangle\} \end{aligned}$$

This relation further indicates that in sector A we have all levels except -1 and in sector B we have all levels except 2.

4.1.2 Phases

A construction project typically involves several *construction phases* such as skeleton, facade, and interior. Each phase has a name and a CU relation which represents the building structure from

the perspective of that phase. For example, the previously given relation $Cu \subseteq \mathit{sector} \times \mathit{level}$ is useful to represent the building for the skeleton phase, where only a coarse granularity is needed. For phases later in the project, such as the facade and interior phases, more fine-grained building structures are needed.

The interior phase concerns the inner structure of the building. A possible way to model it is given by the following CU relation:

$$Cu_{\text{int}} \subseteq \mathit{sector} \times \mathit{level} \times \mathit{section} \times \mathit{number}$$

where attributes *section* and *number* are defined as

$$\begin{aligned} \mathit{section} &:= \{r, k, c\} \\ \mathit{number} &:= \mathbb{N} \end{aligned}$$

Here, *section* describes the technological content of a CU, e.g. whether it is a room (r), a kitchen (k), or a corridor (c). To distinguish several CUs of the same technological content, we assign different numbers to them by means of the attribute *number*. The attributes *sector* and *level* are as defined previously. The room number 5 in the first upper floor in sector A would then be written as $\langle A, 1, r, 5\rangle$.

The facade phase, on the other hand, does not concern inner rooms, but only outside walls of a building. Therefore, a different representation by different attributes is needed. The following CU relation Cu_{fcd} shows one possibility of such a representation:

$$\begin{aligned} \mathit{orientation} &:= \{N, E, S, W\} \\ Cu_{\text{fcd}} &\subseteq \mathit{sector} \times \mathit{orientation} \times \mathit{level} \times \mathit{f\text{-}unit} \end{aligned}$$

Here, an attribute *orientation* is used to define the cardinal direction to which the outside surface of a wall is directed, i.e. north (N), south (S), east (E), or west (W). We can again add more detail by using the *f-unit* attribute, e.g. breaking down walls into small vertical slices. For example, the CU $\langle A, N, 2, 1\rangle$ refers to the first slice on the second level of the north wall of sector A.

Note that in this example, attributes *sector* and *level* are used for all the three phases, while *orientation* and *f-unit* are only used in the facade phase and *section* and *number* are only used in the interior phase. Therefore, a configuration has a global set of available attributes, attribute ranges, and a set of phases. Each phase defines its own CU relation over some subset of the globally available attributes. We illustrate this by summarizing all examples of attributes and phases defined so far into the following configuration:

$$\begin{aligned} \mathit{attributes} &: \{\mathit{sector}, \mathit{orientation}, \mathit{level}, \mathit{section}, \mathit{f\text{-}unit}, \mathit{number}\} \\ \mathit{phases} &: \{Cu_{\text{skl}}, Cu_{\text{fcd}}, Cu_{\text{int}}\} \\ Cu_{\text{skl}} &\subseteq \mathit{sector} \times \mathit{level} \\ Cu_{\text{fcd}} &\subseteq \mathit{sector} \times \mathit{orientation} \times \mathit{level} \times \mathit{f\text{-}unit} \\ Cu_{\text{int}} &\subseteq \mathit{sector} \times \mathit{level} \times \mathit{section} \times \mathit{number} \end{aligned}$$

While this example uses the three phases introduced in the original PRECISE approach, we do not consider these phases to be fixed in our extension. Instead, one is free to define an arbitrary number of phases based on arbitrary attributes. Also note that in the interior and exterior phases, we replaced the *unit* attribute of the original representation with two different attributes *f-unit* and *number*, one for each phase. The reason is that these two attribute represent different concepts: The *f-unit* divides a facade into small units, while the *number* is used to distinguish several CUs of the same *section*.

4.1.3 Craft

A *craft* is a type of workers and is represented by a name only. Examples of crafts are *Plumber*, *Brick Layer*, *Scaffolder* etc. One could add more detailed information to crafts, but for our purpose names are sufficient, as crafts are only used to better describe activities.

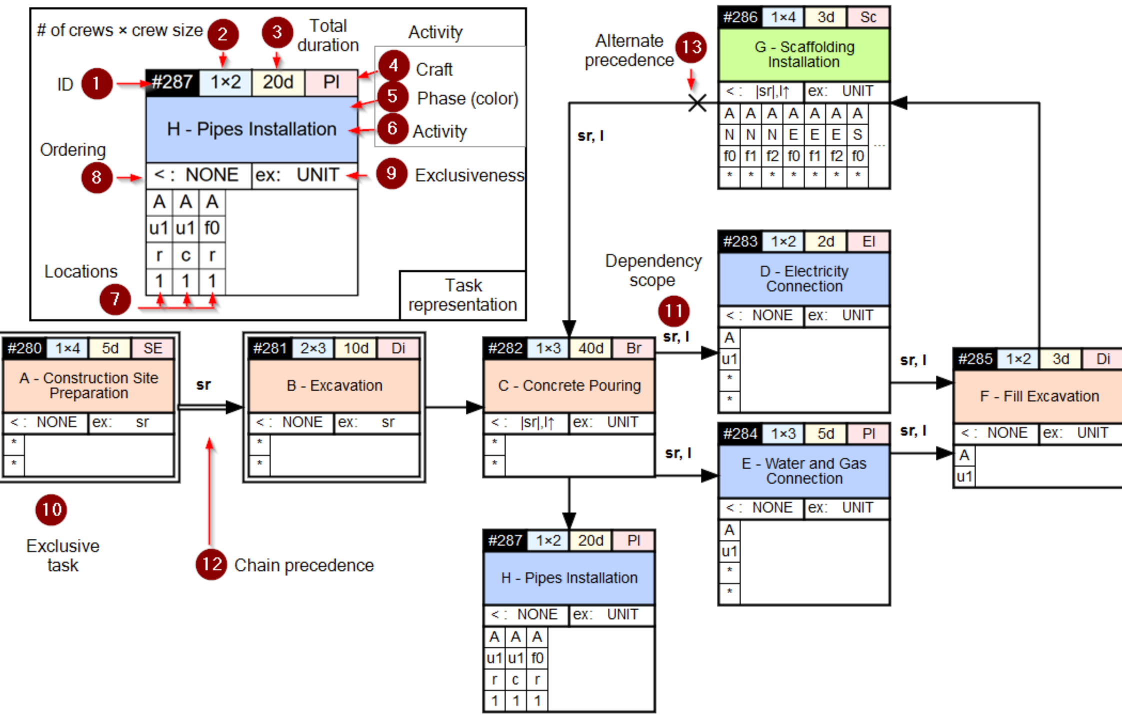


Figure 4.1: Graphical notation of a PRECISE process model. The process model corresponds to the hotel example shown in Fig. 3.4 but uses the improved language.

4.1.4 Activity

Finally, a configuration contains abstract definitions of activities to be performed. An activity is identified by a *name*, and further has *i)* a *phase* to which it belongs, *ii)* a *craft* by which it is to be performed, and *iii)* a *unit of measure* in which the progress of tasks of this kind should be measured compared to a total quantity.

As an example, consider the activity Pipes Installation. It belongs to the interior phase and must be performed by a plumber. Progress can be measured in running meters, representing the total length of pipes which have been installed so far. We define an activity accordingly:

name : Pipes Installation
phase : *interior*
craft : Plumber
unit of measure : running meters (rm)

4.2 Flow Part

The role of the flow part is to model what actually needs to be done in a construction process, using the elements defined in the configuration part. It consists of *tasks* and *dependencies* between tasks. By means of tasks it is possible to define where a certain activity must be performed, while dependencies express precedence relations among them. We define a graphical notation for the flow part, referred to as the *diagram* of a process model. An example of this notation is given in Fig. 4.1. We will explain the concepts and the corresponding notation by means of this example.

4.2.1 Task

In the configuration part, we structured the building into abstract CUs, and defined activities that can be performed in those CUs. In the flow part, we now need to capture in which particular CUs a given activity must be performed. This is the role of the *task*. The basic components of a task are therefore an activity and a set of CUs. Graphically, a task is represented as a box as depicted

in Fig. 4.1 with the fields ① to ⑨. The name of the activity of a task is shown in ④ the second row of the task box. For convenience, also ④ the craft of the activity and ⑤ a color identifying the phase of the activity are shown in a task box. A table at the bottom of a task ⑦ is used to specify the CUs in which the activity is to be performed. Each row represents one attribute of the corresponding CU relation, and CUs are given as columns, by specifying a value for each attribute. Semantically, each task represents the constraint that the corresponding activity must be performed exactly once in each CU listed in the table.

For example, in Fig. 4.1, we represent tasks of the interior by giving them a blue background color. The interior phase has a CU relation $Cu_{int} \subseteq \mathit{sector} \times \mathit{level} \times \mathit{section} \times \mathit{number}$. Given this configuration, the task for activity Pipes Installation, indicates that the activity is to be executed by a plumber in three CUs. They are all contained in sector A, and are composed of a room and a corridor in level $u1$ and another room in level $f0$.

To write CUs in a succinct way, we can use a 'wild-card' symbol, denoted by an asterisk, to refer to multiple CUs at once. Specifically, an asterisk stands for all the values for the given attribute such that there is a corresponding CU. Indeed, it could be the case that all the attributes are specified with an asterisk, meaning that all the CUs for that phase are considered, or only some of the attributes may be specified with an asterisk. In this latter case, all CUs matching the non-asterisk attributes are to be considered. For example, in Fig. 4.1, in the task of activity Electricity Connection, we use wild-cards for *section* and *number* to refer to all CUs in level $u1$ in sector A.

Note that by design, one can define several tasks referring to the same activity. This is useful if different constraints apply to different CUs for the same activity. To distinguish multiple tasks of the same activity, we assign a ① a unique ID to each task. Additionally, a task includes information on the *pitch* of a task, which includes ② the planned number of crews and the optimal crew size, as well as ③ the total duration in days which we expect the crews to spend on this task.

Finally, to control the execution of a task in more detail, it is possible to express some more constraints on how a task is executed in its CUs. These constraints are introduced in the following sections.

4.2.2 Unary Constraints on Tasks

A constraint that applies to a single task is a *unary* constraint. We consider two different types of unary constraints: *ordering* constraints and *exclusiveness* constraints. For simplicity, each task cannot have more than one ordering constraint or more than one exclusiveness constraint. In the diagram, this allows to represent the two types of unary constraints in two fields within the task box; one for ⑧ the ordering constraint and one for ⑨ the exclusiveness constraint.

Ordering

A task with an activity and a set of CUs by itself only requires that the task be executed eventually in all CUs of the given set. The order in which it is executed in its CUs is arbitrary, and it can be executed in multiple CUs at the same time. In practice, however, there are activities where the order among CUs is important. For example, consider the activity Concrete Pouring. We cannot but execute it from bottom to top, starting from the lowest level and proceeding to the next level above whenever one level is finished.

To capture such an ordering constraint from bottom to top, we first model what "bottom to top" means in general. Because this does not depend on a task, we put it into the configuration. In particular, we allow to define a total order on the range of an attribute. For example, suppose the range of *level* is defined as a set of integers from -1 to 2, where each value in the range denotes the offset from the ground level 0. Then the natural order $<$ on integers can be used to define a meaningful total order on the range of *level*.

Once such a total order is defined, the required ordering on Concrete Pouring can be captured by specifying an ⑧ *ordering specification* of $\mathit{level} \uparrow$ to denote that the order in which the task is executed in its CUs must follow the total order defined on *level*. To define a descending order, we write $\mathit{level} \downarrow$, which in the case of *level* is an order from top to bottom. Cleaning of outside walls is an example where such an ordering is useful to ensure that cleaning one level does not dirty other levels below that are already finished.

More complex orderings can be defined by combining multiple attributes. For example, suppose that cleaning must be performed clockwise, e.g. starting on the north side and finishing on the west side, and that each side must be cleaned from top to bottom. To model this, we need an attribute *orientation* with a totally ordered range $\langle N, E, S, W \rangle$. We can use this in an ordering *orientation* $\uparrow$, *level* $\downarrow$ to capture the required constraint.

Note that the position of an attribute in the ordering constraint matters: The ordering specification *level* $\downarrow$, *orientation* $\uparrow$ specifies a different constraint, namely that levels must be cleaned from top to bottom, and each level is cleaned clockwise, starting at the north side and proceeding clockwise until the south side is finished. Thus, in an ordering composed of multiple attributes, the order of an individual attribute only applies to groups of CUs that share the same value for all attributes on the left.

This concept of partitioning is also useful to define orderings that allow some degree of parallelism. For example, consider a project with two separate buildings A and B, represented by an attribute *sector*. In each building, the task Concrete Pouring must be performed from bottom to top as usual. But when the first level of building A is finished, Concrete Pouring can already start in the next level of A, regardless of whether the first level of building B is already finished or not. Hence, the task can be executed independently in the two buildings. To capture such a constraint, we write an ordering $|sector|, level \uparrow$. The first part $|sector|$ partitions the CUs into groups of CUs of the same building, before the second part *level* $\uparrow$ is applied to each group.

We now define orderings in more general terms. An ordering is given as a sequence $\mathcal{O} = \langle \mathsf{o}_1\alpha_1, \ldots, \mathsf{o}_n\alpha_n \rangle$, where each $\mathsf{o}_i\alpha_i$ is an *order expression*, formed by applying an *ordering operator* o_i to an attribute α_i. Each expression $\mathsf{o}_i\alpha_i$ partitions the given CU relation into groups of the same values for the attribute α_i. Depending on the particular operator o_i, an ordering on the resulting groups is specified. There are three available ordering operators:

- $\alpha_i \uparrow$: Partition CUs by attribute α_i, and order the resulting groups *ascending* according to the total order on the range of attribute $attr_i$.
- $\alpha_i \downarrow$: Partition CUs by attribute α_i, and order the resulting groups *descending* according to the total order on the range of attribute α_i.
- $|\alpha_i|$: Only partition the CUs by attribute α_i without ordering the resulting groups.

To order one group of CUs before another group of CUs, means to order all CUs contained in the former before all CUs contained in the latter.

To apply an ordering $\mathcal{O} = \langle \mathsf{o}_1\alpha_1, \ldots, \mathsf{o}_n\alpha_n \rangle$ to a given CU relation Cu, we read $\mathcal{O}$ from left to right. Starting with the left-most expression $\mathsf{o}_1\alpha_1$, we partition Cu into several groups such that all the CUs of one group have the same value for α_1. If o_1 requires an ordering on these groups, we apply it. Then, we apply the next expression $\mathsf{o}_2\alpha_2$ to each group resulting from the previous expression. That is, each group corresponding to one value for attribute α_1 is further partitioned into subgroubs by the attribute α_2. These resulting subgroubs of the given group are then ordered according to operator o_2. We continue in a similar way for the remaining attributes until all expressions are evaulated.

Thus, the semantics of applying an ordering $\mathcal{O} = \langle \mathsf{o}_1\alpha_1, \ldots, \mathsf{o}_n\alpha_n \rangle$ to a given CU relation Cu is defined recursively. First, we apply expression $\mathsf{o}_1\alpha_1$ to Cu and obtain a partition of Cu by α_1. Then, each expression $\mathsf{o}_{i+1}\alpha_{i+1}$ is applied to each group resulting from the previous expression $\mathsf{o}_i\alpha_i$. In this way, the ordering $\mathcal{O}$ defines a partial order $<_{\mathcal{O},Cu}$ on the elements of Cu.

Let us look at some examples. Consider the following CU relation

$$Cu_{skl} \subseteq sector \times level$$
$$Cu_{skl} = \{\langle A, 0 \rangle, \langle A, 1 \rangle, \langle B, 0 \rangle, \langle B, 1 \rangle\}$$

Assume that sectors are ordered alphabetically, and levels are ordered naturally, i.e. $A < B$ and $0 < 1$. Applying the ordering $\mathcal{O}_1 = \mathit{level} \uparrow$ to Cu results in the following partial order $<_{\mathcal{O}_1,Cu}$:

$$\langle A, 0\rangle <_{\mathcal{O}_1,Cu} \langle A, 1\rangle$$
$$\langle A, 0\rangle <_{\mathcal{O}_1,Cu} \langle B, 1\rangle$$
$$\langle B, 0\rangle <_{\mathcal{O}_1,Cu} \langle A, 1\rangle$$
$$\langle B, 0\rangle <_{\mathcal{O}_1,Cu} \langle B, 1\rangle$$

That is, all CUs in level 0 come before all CUs of level 1, regardless of the sector. If instead we want to apply the ascending order on the level only within each sector, we can define the ordering $\mathcal{O}_2 = |\mathit{sector}|, \mathit{level} \uparrow$. This results in the following partial order $<_{\mathcal{O}_2,Cu}$:

$$\langle A, 0\rangle <_{\mathcal{O}_2,Cu} \langle A, 1\rangle$$
$$\langle B, 0\rangle <_{\mathcal{O}_2,Cu} \langle B, 1\rangle$$

Another possibility is to require an ascending order on levels, and within each level a descending order on sectors. To capture this, we write an ordering $\mathcal{O}_3 = \mathit{level} \uparrow, \mathit{sector} \downarrow$, which results in the partial order $<_{o3,Cu}$ defined as:

$$\langle B, 0\rangle <_{\mathcal{O}_3,Cu} \langle A, 0\rangle <_{\mathcal{O}_3,Cu} \langle B, 1\rangle <_{\mathcal{O}_3,Cu} \langle A, 1\rangle$$

Exclusiveness

In a construction project, multiple crews perform multiple activities on the construction side. To avoid interference, we forbid the execution of more than one activity in a single CU at the same time. That is, every crew is granted *exclusive* access to a CU as long as it performs an activity there.

For certain critical activities, however, this is not enough to prevent interference. As an example, suppose that when windows are installed in a floor, some windows lay on the ground in the corridor. To prevent damage, one could decide that the task Window Installation needs exclusive access to the whole level. Or, maybe it is enough to require exclusive access to the level within a sector. In general, there is the need to refer to coarse-grained areas of a building, which may contain several CUs. Such areas are called *Construction Areas* (CAs). For that purpose, we define the *scope*, which is a set of attributes and specifies a granularity of CAs. At the scope *level*, for example, we have a CA for each level in the project. Each of these CAs contains all CUs in the corresponding level. A scope $\mathit{sector}, \mathit{level}$ instead results in one CA per combination of sector and level. For a given CU relation $Cu \subseteq \alpha_1 \times \cdots \times \alpha_n$, the finest possible scope is the *unit* scope, which consists of all attributes $\alpha_1, \ldots, \alpha_n$ and yields a CA for each CU. The coarsest possible scope instead is the *global* scope $\varnothing$, and results in one single CA, containing all $\mathsf{u} \in Cu$.

We use scopes to capture ⑨ *exclusiveness constraints.* Given a task, the scope of the exclusiveness constraint specifies the granularity of the CAs to which the task is granted exclusive access. More precisely, for a task t with CUs Cu and an exclusiveness constraint at scope s, for every other task t' with CUs Cu', for every CU $\mathsf{u}' \in Cu'$, task t' must be executed in u' either before or after all CUs $\mathsf{u} \in Cu$ such that u and u' have the same values for all attributes in s. Note that all attributes in the scope of the exclusiveness constraint must be contained in the phase to which the task belongs. By default, a task has an *implicit exclusiveness* exclusiveness constraint at the unit scope. This captures the basic requirement that the task has exclusive access to single CUs. To give the task exclusive access to wider areas, we can coarsen the scope by removing attributes, resulting in an *explicit exclusiveness.* One can also give a task exclusive access to the whole construction site by using an exclusiveness constraint at the global scope $\varnothing$. This means that while the given task is being performed, other tasks cannot be executed in any CU until the exclusive task is finished completely.

In the diagram, a ⑩ double border around a box highlights tasks with explicit exclusiveness in the diagrams. As an example, consider the task Excavation in Fig. 4.1, which belongs to the skeleton phase. CUs of this phase are defined in terms of the two attributes *sector* and *level*. By default, the task would have an exclusiveness at the unit scope $\mathit{sector}, \mathit{level}$. In that case, whenever it is executed in a CU, i.e. in a certain level and sector, no other tasks can be executed there until

Excavation is finished. This is the least restrictive exclusiveness constraint that we can have for this task. In the diagram, however, the exclusiveness constraint of this task has the scope *sector*. This means that whenever the task starts in a CU, it has exclusive access to the whole sector in which that CU is contained until it is finished in all CUs of that sector.

4.2.3 Dependencies between two Tasks

As opposed to *unary* constraints, a process model can also have *binary* constraints, called *dependencies*. In particular, a dependency relates a *source* task to a *target* task, and applies at a granularity specified by a scope. In the diagram, a dependency is represented as an arrow from the source task to the target task. The scope of the dependency is displayed as a label on the arrow. To distinguish the different kinds of dependencies in the diagram, we vary the presentation of the arrow.

Basic Precedence

In a construction project, it is very common to have precedences between two tasks, i.e. that one task cannot start until another task is finished. As an example, consider the two tasks Excavation and Concrete Pouring in Fig. 4.1. We cannot pour the concrete before excavation finished. In particular, the excavation must be finished in all CUs before the concrete can be poured in any CUs. Differently, we can also have precedences within CAs of particular scopes. Consider the tasks Concrete Pouring and Electricity Connection in Fig. 4.1 as an example. On the one hand, the poured concrete is needed to install electricity in a CU. On the other hand, it is not necessary to wait until the concrete is poured everywhere before starting with the electricity installation in the first CU.

Thus, we must be able to capture precedence constraints at various granularities of CUs. For that purpose, we introduce the *basic precedence* from a *source* task to a *target* task at a given *scope*. The scope is again specified as a set of attributes and determines a granularity of CAs whithin which the precedence applies. More precisely, the execution of the source task in a CU must precede the execution of the target task in a CU iff the two CUs share the same values for all attributes in the scope, i.e. iff the two CUs are contained in the same CA at the given scope. Note that this implies that the precedence only applies to shared CAs.

In the diagram, a basic precedence is depicted as an arrow with a single solid line from the source task to the target task, and the (11) scope is denoted as a label on the arrow. For example, the arrow from Concrete Pouring to Electricity Connection in Fig. 4.1 is labeled with the scope sr, l as an abbreviation for $sector, level$ to indicate that the dependency applies to each level in each sector. An arrow without a label represents a basic precedence at the global scope $\varnothing$. An example in Fig. 4.1 is the arrow from Excavation to Concrete Pouring.

Note that the source and target of a basic precedence can be of different phases. In that case, all attributes the scope of the precedence must be contained in both phases, i.e. the scope must be a subset of the intersection of the set of attributes of the two phases. For this reason, attributes should be defined in such a way that they can be interpreted independently of the phase in which it is used.

As an example, consider the attributes *f-unit* in the exterior phase and the *number* in the interior phase. They both have a range from 1 to n, so technically, we could could replace them with a single attribute *unit* with such a range, which is how these two phases were represented in the original PRECISE approach. But once the attribute is shared between the two phases, we can use it in the scope of a basic precedence from a task in one phase to a task in the other phase. For example, if we replace the attributes with *unit*, we could define a basic dependency from Electricity Connection to Scaffolding Installation at a scope *unit*. This means that we cannot install the scaffolding in a facade unit of number 1 until we finished Electricity Connection in all CUs with a room number 1. Defining such precedence relations between actually unrelated CUs is not what the scope was designed for. Instead, the idea of the scope is to specify a granularity of areas where one task must be finished completely before the other one can start.

The remaining kinds of dependencies are refinements of the basic precedence. That is, they take the basic precedence as a starting point and add more constraints.

Chain Precedence

Suppose that wooden windows are used in a project. To avoid damages, the windows must be covered after their installation, and no other tasks can be executed between the installation and the covering. We introduce chain precedences to capture constraints of this kind.

Just like a basic precedence, a chain precedence has a source task, a target task, and a scope. The semantics of an alternate precedence is defined by the semantics of the corresponding basic precedence plus an additional constraint: Together, the source and the target task must have exclusive access to CAs at the scope of the chain precedence. That is, once the source task starts in a CA at the chain precedence's scope, no tasks other than the source and the target can be executed in that CA until the target (and the source) task is finished in all CUs in that CA.

Graphically, a chain precedence is denoted by an arrow with a double line to indicate the additional constraint which is a kind of exclusiveness constraint on the two tasks. As an example, consider Fig. 4.1 and the chain precedence from Construction Site Preparation to Excavation at scope *sector*. It requires that once we start with Construction Site Preparation in some CU, we need to finish both Construction Site Preparation and Excavation in the sector in which this CU is contained before we can execute any tasks other than these two in that sector.

Note that in this example, the chain precedence is redundant with respect to a basic precedence. It would not change anything if we would replace the chain precedence with a basic precedence, because anyway all other tasks need to wait until both, Construction Site Preparation and Excavation, are finished completely due to other dependencies. Still, one may decide to put this constraint nevertheless to make the diagram more robust. For example, if we are not sure whether some dependencies will be removed or new tasks will be introduced later on, it might be better to put the constraint explicitly.

Alternate Precedence

Consider the two tasks Scaffolding Installation and Concrete Pouring. Both must be performed from bottom to top in a building, and in each level, the scaffolding must be installed before the concrete can be poured. These constraints can be captured by putting an ascending ordering by *level* on the two tasks and adding a dependency at the scope *level* from Scaffolding Installation to Concrete Pouring. There is, however, one more constraint to be captured: When the scaffolding is installed in one level, also the concrete must be poured in that level before the scaffolding can be installed in the level above. To capture these kinds of constraints, we introduce alternate precedences.

Similar to chain precedences, an alternate precedence is also a basic precedence from a source task to a a target task at some scope with an additional constraint. This additional constraint requires that the two tasks can be executed in at most one CA at the given scope at a time. That is, when the source task starts in some CU `u`, it cannot be performed in any CA at the given scope except in the one where `u` is contained until also the target task is finished in that CA.

In the diagram, an (12) alternate precedence is denoted as a dependency with a cross near the source task to indicate the additional waiting constraint on the source task. Figure 4.1 provides an example of an alternate precedence from Scaffolding Installation to Concrete Pouring at scope *sector*, *level*. On the one hand, this requires that in every level of every sector, we must first finish Scaffolding Installation before we can start Concrete Pouring, because every alternate precedence is a dependency. On the other hand, this further requires that once we start installing the scaffolding at a level in a sector, we must wait until we finished pouring the concrete in the same CA until we can start to install the scaffolding at another level or in another section.

4.3 Formal Semantics

We now introduce formal semantics for the various constraints in a process model to eliminate the ambiguity in the original PRECISE process modeling language. For that purpose, we use Linear Temporal Logic. We will first give a short introduction into LTL before actually defining the various types of constraints in terms of LTL. On this basis, we then define the consistency of a model and conformance of a schedule to a model.

4.3.1 Linear Temporal Logic

The semantics of a process model is defined in terms of Linear Temporal Logic (LTL). In the following, we briefly recap the basics of LTL. For more more details see (Baier and Katoen, 2008; Clarke et al., 2001).

As in propositional logic, LTL formulas are built by connecting atomic propositions by logical connectives. Yet, in LTL we assume that the truth evaluations of these propositions may change over time. Therefore, in LTL we have propositional operators as well as temporal operators. The following defines the grammar of an LTL formula, where p is a propositional atom:

$$\begin{aligned}\phi ::= & \top \mid \bot \mid p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \rightarrow \phi \mid (\phi) \mid \\ & \bigcirc\phi \mid \Diamond\phi \mid \Box\phi \mid \phi \mathsf{U} \phi\end{aligned}$$

The temporal operators are displayed in the second line. The intuition of their semantics can be described in the following way:

– $\bigcirc\phi$: Formula ϕ holds in the *next* state.

– $\Diamond\phi$: Formula ϕ *eventually* holds, i.e. either now or in some future state.

– $\Box\phi$: Formula ϕ *always* holds, i.e. now and in all future states.

– $\phi \mathsf{U} \psi$: Formula ϕ holds *until* formula ψ holds.

4.3.2 Events

LTL formulas are based on atomic propositions. Therefore, we need to define the set of atomic propositions for a process model before we can describe any constraint in terms of LTL. Such atomic propositions should provide all information needed to evaluate all possible constraints in a model. At this point, note that the execution of an activity in a CU is considered not to be atomic. Instead, the execution starts at some point in time, and has a certain duration until it ends later. It is also possible that multiple activities are being executed simultaneously (in different CUs). The various types of constraints that we have are of the kind "end-to-start", meaning that they rule when an activity can start in a CU depending on whether another activity has ended in a CU. It is therefore natural to use *start* and *end events* of tasks in CUs as atomic propositions. More precisely, for every task of activity T and of CUs Cu, and for each $\mathsf{u} \in Cu$, the set of atomic propositions AP contains a start event $\mathit{start}(T, \mathsf{u})$ and an end event $\mathit{end}(T, \mathsf{u})$. We will use the term *event* to refer to an atomic proposition.

For example, consider a model with a single task in the skeleton phase defined as follows.

$$\begin{aligned}\mathit{activity} &: \textsc{Concrete Pouring} \\ \mathit{CUs} &: \{\langle A, 0\rangle, \langle A, 1\rangle, \langle B, 0\rangle\}\end{aligned}$$

This results in the following set of atomic propositions AP, where $T = \textsc{Concrete Pouring}$:

$$\begin{aligned}AP = \{&\mathit{start}(T, \langle A, 0\rangle), \mathit{end}(T, \langle A, 0\rangle), \\ &\mathit{start}(T, \langle A, 1\rangle), \mathit{end}(T, \langle A, 1\rangle), \\ &\mathit{start}(T, \langle B, 0\rangle), \mathit{end}(T, \langle B, 0\rangle)\}\end{aligned}$$

4.3.3 Constraints

In this section, we provide formal semantics of all types of constraints that we can have in a process model in terms of LTL formulas. But first, we introduce some auxiliary notation for being able to write the formulas in a more concise and convenient way. These notations are *projections* and *selections* over CUs and are inspired by relational algebra.

Projection We already gave an intuition on how to obtain CAs of a granularity coarser than that of a given CU relation. Now we define the *projection* of a CU to a scope for being able to refer to CAs more formally and concisely.

Let $Cu \subseteq \alpha_1 \times \cdots \times \alpha_n$ be a CU relation, let $\mathsf{u} \in Cu$ be a CU, and let $\mathsf{s} \subseteq \{\alpha_1, \ldots, \alpha_n\}$ be a *scope* of attributes in Cu. The *projection* of u to s is defined as

$$\Pi_{\mathsf{s}}(\mathsf{u}) := \pi_{\mathsf{s}}$$

where π_{s} is the CA at scope s that contains u. More precisely, π_{s} is a tuple $\langle v'_1, \ldots, v'_m \rangle$ that contains only those values of u that correspond to an attribute that is contained in s, whereas the values for attributes that are not in s are ignored.

For instance, given a CU relation $Cu \subseteq \mathit{sector} \times \mathit{level} \times \mathit{section} \times \mathit{number}$, and a CU $\langle A, 0, r, 1 \rangle \in Cu$, we can define the following projections:

$$\begin{aligned}
\Pi_{\mathit{sector},\mathit{level},\mathit{section},\mathit{number}}(\langle A, 0, r, 1 \rangle) &= \langle A, 0, r, 1 \rangle \\
\Pi_{\mathit{sector},\mathit{level},\mathit{section}}(\langle A, 0, r, 1 \rangle) &= \langle A, 0, r \rangle \\
\Pi_{\mathit{level},\mathit{section}}(\langle A, 0, r, 1 \rangle) &= \langle 0, r \rangle \\
\Pi_{\mathit{level}}(\langle A, 0, r, 1 \rangle) &= \langle 0 \rangle \\
\Pi_{\varnothing}(\langle A, 0, r, 1 \rangle) &= \langle \rangle
\end{aligned}$$

Moreover, we also allow to apply projections to sets of CUs. The projection of a set of CUs Cu to a scope s is defined as the set of the projections of each CU in Cu to s, i.e.

$$\Pi_{\mathsf{s}}(Cu) := \{\Pi_{\mathsf{s}}(\mathsf{u}) \mid \mathsf{u} \in Cu\}$$

Selection Similarly, in some cases it is useful to select those CUs of a given set that are contained in a given CA of higher scope. This is called a *selection*. Given a CU relation Cu and a CA π_{s} of scope s, the selection of the CUs in Cu that are contained in π is defined as

$$\sigma_{\pi_{\mathsf{s}}}(Cu) := \{\mathsf{u} \in Cu \mid \Pi_{\mathsf{s}}(\mathsf{u}) = \pi_{\mathsf{s}}\}$$

As an example, consider the following CU relation:

$$\begin{aligned}
&Cu \subseteq \mathit{sector} \times \mathit{level} \times \mathit{section} \times \mathit{number} \\
&Cu = \{\langle A, 0, r, 1 \rangle, \langle A, 0, b, 1 \rangle, \langle A, 1, r, 1 \rangle, \langle B, 0, r, 1 \rangle\}
\end{aligned}$$

Suppose we want to select all CUs at level 0 of sector A. This corresponds to the CUs contained in a CA of scope *sector*, *level* defined as $\pi_{\mathit{sector},\mathit{level}} = \langle A, 0 \rangle$. We can use a selection by $\pi_{\mathit{sector},\mathit{level}}$ to obtain the desired result:

$$\sigma_{\pi_{\mathit{sector},\mathit{level}}}(Cu) = \{\langle A, 0, r, 1 \rangle, \langle A, 0, b, 1 \rangle\}$$

Similarly, if we are interested in those CUs that are actually rooms, we use a CA at scope *section* defined as $\pi_{\mathit{section}} = \langle r \rangle$ to select them:

$$\sigma_{\pi_{\mathit{section}}}(Cu) = \{\langle A, 0, r, 1 \rangle, \langle B, 0, r, 1 \rangle, \langle A, 2, r, 1 \rangle\}$$

Now we are ready to define LTL formulas for all types of constraints that we can have in a model. Table 4.2 provides the LTL formulas for existence constraints, whereas formulas for the various kinds of dependencies are shown in Table 4.3. Because some subformulas are recurring in various constraints, we defined them separately in Table 4.1. In these tables, a task is represented as a pair $\langle T, Cu \rangle$, where T is an activity and Cu is the set of CUs where T is required to be performed by the given task. In particular, for brevity, every pair of an activity T and a set of Cu is called a task, even if it does not correspond to a task in the original model, because it is of the same form.

4.3.4 Execution

To understand how LTL formulas are interpreted, let us look at propositional logic first. In propositional logic, formulas are interpreted over valuations of atomic propositions. A valuation is a mapping from each atomic proposition to the values true or false. One can evaluate the truth

End-to-start: *end_to_start*($\langle \mathsf{a}, Cu_\mathsf{a}\rangle$:*task*, $\langle \mathsf{b}, Cu_\mathsf{b}\rangle$:*task*)
Activity b cannot start in any $\mathsf{u_b} \in Cu_\mathsf{b}$ until activity a is finished in all $\mathsf{u_a} \in Cu_\mathsf{a}$. $$\forall \mathsf{u_a} \in Cu_\mathsf{a}, \mathsf{u_b} \in Cu_\mathsf{b}\ \neg \textit{start}(\mathsf{b}, \mathsf{c_b})\ \mathsf{U}\ \textit{end}(\mathsf{a}, \mathsf{c_a})$$
Uninterrupted: *uninterrupted*(τ_1:*tasks*, τ_2:*tasks*)
All tasks in τ_1 must be executed in all of their CUs without being interrupted by the execution of any task in τ_2 in any of their CUs and vice-versa. $$\forall \langle \mathsf{a}_1, Cu_1\rangle \in \tau_1,\ \langle \mathsf{a}_2, Cu_2\rangle \in \tau_2$$ $$\text{end_to_start}(\mathsf{a}_1, Cu_1, \mathsf{a}_2, Cu_2)$$ $$\vee$$ $$\forall \langle \mathsf{a}_1, Cu_1\rangle \in \tau_1,\ \langle \mathsf{a}_2, Cu_2\rangle \in \tau_2$$ $$\text{end_to_start}(\mathsf{a}_2, Cu_2, \mathsf{a}_1, Cu_1)$$

Table 4.1: Macros for subformulas recurring in formulas of several constraints.

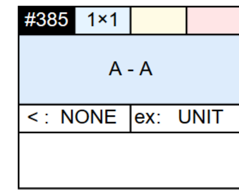
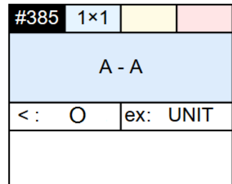
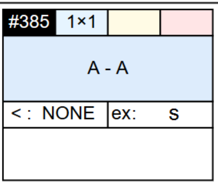
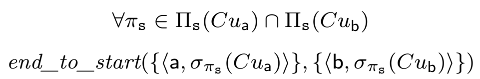
<table>
<tr><td colspan="2">EXISTENCE: existence($\langle \mathsf{a}, Cu_\mathsf{a}\rangle$:task)
Activity a must be executed in all CUs in Cu_a. Further, while a is executed in a CU $\mathsf{u_a} \in Cu_\mathsf{a}$, no other activities can be executed in $\mathsf{u_a}$ until a is finished there.</td></tr>
<tr><td>$$\forall \mathsf{u_a} \in Cu_\mathsf{a}$$ $$\Diamond \textit{start}(\mathsf{a}, \mathsf{u_a}) \wedge \Diamond \textit{end}(\mathsf{a}, \mathsf{u_a})$$ $$\wedge \forall \langle \mathsf{b}, Cu_\mathsf{b}\rangle,\ \mathsf{u_b} \in \sigma_\mathsf{u}(Cu_B) \text{ s.t. } \mathsf{a} = \mathsf{b} \rightarrow \mathsf{u_b} \notin Cu_\mathsf{a}$$ $$\textit{uninterrupted}\langle \mathsf{a}, \{\mathsf{u_a}\}\rangle, \langle \mathsf{b}, \{\mathsf{u_b}\}\rangle$$</td><td></td></tr>
<tr><td colspan="2">ORDERED EXISTENCE: ordered_existence($\langle \mathsf{a}, Cu\rangle$:task, $\mathcal{O}$:ordering)
Activity a must be executed in all construction areas in Cu following the ordering Cu, which defines a partial order $<_{\mathcal{O},Cu}$ over Cu.</td></tr>
<tr><td>$$\textit{existence}(\mathsf{a}, Cu) \wedge$$ $$\forall \mathsf{u}_1, \mathsf{u}_2 \in Cu \text{ s.t. } \mathsf{u}_1 <_{\mathcal{O},Cu} \mathsf{u}_2$$ $$\textit{end_to_start}(\{\mathsf{a}, \mathsf{u}_1\}, \{\mathsf{a}, \mathsf{u}_2\})$$</td><td></td></tr>
<tr><td colspan="2">EXCLUSIVE EXISTENCE: exclusive_existence($\mathsf{t} = \langle \mathsf{a}, Cu_\mathsf{a}\rangle$:task, s:scope)
As long as activity a is executed in some construction areas in Cu_a, no other activity can be executed in any construction area in Cu_a.</td></tr>
<tr><td>$$\textit{existence}(\mathsf{a}, Cu_\mathsf{a}) \wedge$$ $$\forall \mathsf{c_s} \in \Pi_\mathsf{s}(Cu_\mathsf{a}),\ \forall \langle \mathsf{b}, Cu_\mathsf{b}\rangle,\ \mathsf{u_b} \in \sigma_{\pi_\mathsf{s}}(Cu_\mathsf{b}),$$ $$\text{s.t. } \mathsf{a} = \mathsf{b} \rightarrow \mathsf{u_b} \notin Cu_\mathsf{a}$$ $$\textit{uninterrupted}(\{\langle \mathsf{a}, \sigma_{\pi_\mathsf{s}}(Cu_\mathsf{a})\rangle\}, \{\langle \mathsf{b}, \{\mathsf{u_b}\}\rangle\})$$</td><td></td></tr>
</table>

Table 4.2: LTL formulas for existence constraints.

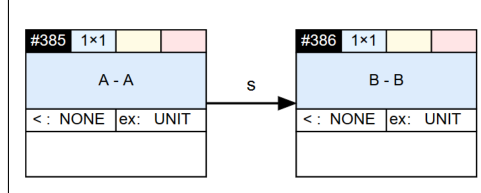
<u>Basic Precedence:</u> *precedence*($\langle \mathsf{a}, Cu_{\mathsf{a}} \rangle$:*task*, $\langle \mathsf{b}, Cu_{\mathsf{b}} \rangle$:*task*, s:*scope*)

In each CA at scope s that is shared between Cu_{a} and Cu_{b}, activity b cannot start until a is finished.



$$\forall \pi_{\mathsf{s}} \in \Pi_{\mathsf{s}}(Cu_{\mathsf{a}}) \cap \Pi_{\mathsf{s}}(Cu_{\mathsf{b}})$$

$$\mathit{end_to_start}(\{\langle \mathsf{a}, \sigma_{\pi_{\mathsf{s}}}(Cu_{\mathsf{a}}) \rangle\}, \{\langle \mathsf{b}, \sigma_{\pi_{\mathsf{s}}}(Cu_{\mathsf{b}}) \rangle\})$$

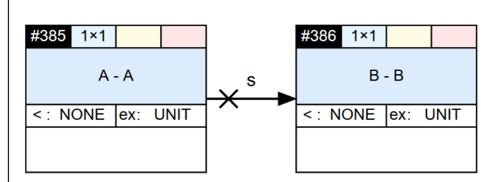


<u>Alternating Precedence:</u> *alternate_precedence*($\langle \mathsf{a}, Cu_{\mathsf{a}} \rangle$:*task*, $\langle \mathsf{b}, Cu_{\mathsf{b}} \rangle$:*task*, s:*scope*)

In each CA at scope s that is shared between Cu_{a} and Cu_{b}, activity b cannot start until a is finished. Also, while one of *actA* and *actB* are performed, none of the two can be performed in another shared CA at the same scope s.

$$(\mathsf{a} = \mathsf{b} \vee \mathit{precedence}(\mathsf{a}, Cu_{\mathsf{a}}, \mathsf{b}, Cu_{\mathsf{b}}, \mathsf{s})) \wedge$$
$$\forall \pi_{\mathsf{s}}, \pi'_{\mathsf{s}} \in \Pi_{\mathsf{s}}(Cu_{\mathsf{a}}) \cap \Pi_{\mathsf{s}}(Cu_{\mathsf{b}}), \text{ s.t. } \pi \neq \pi'$$

$$\mathit{uninterrupted}(\{\langle \mathsf{a}, \sigma_{\pi_{\mathsf{s}}}(Cu_{\mathsf{a}}) \rangle, \langle \mathsf{b}, \sigma_{\pi_{\mathsf{s}}}(Cu_{\mathsf{b}}) \rangle\},$$
$$\{\langle \mathsf{a}, \sigma_{\pi'_{\mathsf{s}}}(Cu_{\mathsf{a}}) \rangle, \langle \mathsf{b}, \sigma_{\pi'_{\mathsf{s}}}(Cu_{\mathsf{b}}) \rangle\},)$$



<u>Chain Precedence:</u> *chain_precedence*($\langle \mathsf{a}, Cu_{\mathsf{a}} \rangle$:*task*, $\langle \mathsf{b}, Cu_{\mathsf{b}} \rangle$:*task*, s:*scope*)

In each CA at scope s that is shared between Cu_{a} and Cu_{b}, activity b cannot start until a is finished. Also, while one of *actA* and *actB* are performed in a CA at scope s (not necessarily shared), other tasks cannot be performed in the same CA.

$$\mathit{precedence}(\mathsf{a}, Cu_{\mathsf{a}}, \mathsf{b}, Cu_{\mathsf{b}}, \mathsf{s}) \wedge \forall \pi_{\mathsf{s}} \in \Pi_{\mathsf{s}}(Cu_{\mathsf{a}}) \cup \Pi_{\mathsf{s}}(Cu_{\mathsf{b}}),$$
$$\forall \langle \mathsf{c}, Cu_{\mathsf{c}} \rangle,\ \mathsf{u_c} \in \sigma_{\pi_{\mathsf{s}}}(Cu_{\mathsf{c}}) \text{ s.t. } \mathsf{a} = \mathsf{c} \rightarrow \mathsf{u_c} \notin Cu_{\mathsf{a}}, \text{ if}$$
$$\mathsf{b} = \mathsf{c} \rightarrow \mathsf{u_c} \notin Cu_{\mathsf{b}}$$

$$\mathit{uninterrupted}(\{\langle \mathsf{a}, \sigma_{\pi_{\mathsf{s}}}(Cu_{\mathsf{a}}) \rangle, \langle \mathsf{b}, \sigma_{\pi_{\mathsf{s}}}(Cu_{\mathsf{b}}) \rangle\},$$
$$\{\langle \mathsf{c}, \{\mathsf{u_c}\} \rangle\},)$$

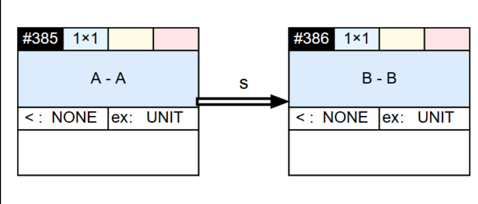


Table 4.3: LTL formulas for dependency constraints.

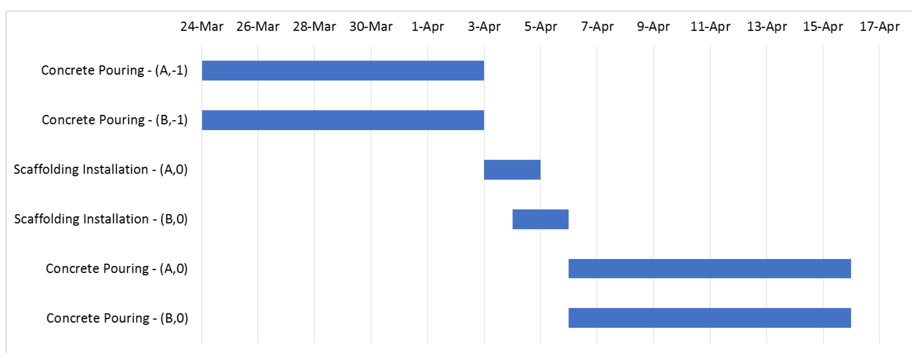


Figure 4.2: Gantt chart representing a schedule of the activities Scaffolding Installation and Concrete Pouring in two sectors A and B, where A has levels from -1 to 1 and B has levels -1 and 0.

value of the whole formula by replacing each atomic proposition with the truth value assigned by the valuation and applying the semantics of the connectives.

Temporal logics such as LTL also consider valuations of atomic propositions, but assume that truth assignments of atomic propositions may change over time. In particular, LTL has a *linear* model of time. That is, we consider one possible evolvement of truth assignments over time, and check if it satisfies the given LTL formula. To represent the change of truth assignments of a set of atomic propositions AP over time, we use an infinite sequences of states $\sigma = s_0, s_1, \ldots$. Each state represents one point in time and contains those atomic propositions of AP that evaluate to true at that time.

In our context of process modeling in construction, atomic propositions are start end end events of activities in CUs. Therefore, given a process model $\mathcal{M}$, an infinite sequence of states over events represents an *execution* of $\mathcal{M}$. Such an execution indicates in which order activities are executed in CUs. Multiple events may happen at the same time, and multiple activities may be performed in different CUs at the same time.

Schedules In particular, for two given events, an execution only captures which one occurs first and which second, or whether both occur simultaneously. It does not capture the concrete time at which an event occurs, nor the actual time that passes between two events. While such information about time is not needed for checking whether an execution satisfies all constraints, it is necessary for actually carrying out the work. Therefore, projects are typically planned using *schedules*, where each piece of work has a start date, an and date, and a duration. Traditionally, pieces of work are activities, whereas in our context, they are a combination of an activity and a CU.

A popular graphical representation of schedules is the Gantt chart. Activities (and CUs) are listed on a vertical axis, and the planned execution of each activity is plotted as a bar on the horizontal time axis. Figure 4.2 shows a Gantt chart representing a schedule of two activities in multiple CUs.

To interpret the LTL formulas of the constraints in a model over a Gantt chart, we can translate the Gantt chart into an execution. This can be done by starting at the first date in the Gantt chart and go forward in time until the end of the project. For each date, we collect all occurring events in a new state and add it to the resulting sequence. For an activity a and a CU u, the start of a in u occurs on a given date d iff d is the first date where a is scheduled in u. Similarly, the end of an activity a in a CU u occurs on a given date d iff d is the first date where a is not scheduled in u anymore, i.e. iff a is not scheduled in u at d but was scheduled at $d-1$. If no event occurs at a particular day, we can avoid adding a state. After the last date in the project, we also consider the next day for getting the missing end events. Then, no further events will occur. This corresponds to another state with an empty set of events that is repeated infinitely often. The result is an infinite sequence of states that can be checked against constraints expressed in LTL.

As an example, consider the Gantt chart shown in Fig. 4.2. Suppose we have a model with an alternate precedence from Scaffolding Installation to Concrete Pouring at the scope of *level*. To use LTL semantics for checking whether the Gantt chart satisfies this constraint, we

first translate the Gantt chart to an execution as described previously. The following sequence of states represent the resulting execution, where CP denotes the activity CONCRETE POURING and SI denotes the activity SCAFFOLDING INSTALLATION.

$$\begin{aligned}
&\{\mathit{start}(CP, \langle A, -1\rangle), \mathit{start}(CP, \langle B, -1\rangle)\},\\
&\{\mathit{end}(CP, \langle A, -1\rangle), \mathit{end}(CP, \langle B, -1\rangle), \mathit{start}(SI, \langle A, 0\rangle)\},\\
&\{\mathit{start}(SI, \langle B, 0\rangle\},\\
&\{\mathit{end}(SI, \langle A, 0\rangle\},\\
&\{\mathit{end}(SI, \langle B, 0\rangle), \mathit{start}(CP, \langle A, 0\rangle), \mathit{start}(CP, \langle B, 0\rangle)\},\\
&\{\mathit{end}(CP, \langle A, 0\rangle), \mathit{end}(CP, \langle B, 0\rangle)\}
\end{aligned}$$

Then we apply standard LTL semantics over the resulting execution and conclude that the execution, and therefore also the original Gantt chart, indeed satisfies the constraint.

Well-defined Execution Note that independently of the constraints in a particular model, executions must follow certain rules in order to be feasible. One such rule is that an activity can only end in a particular CU if it has started there already before. Second, we assume that each activity is performed at most once in a CU. In a process model, and specifically in the LTL constraints shown in Tables 4.2 and 4.2, we only consider *well-defined* executions, i.e. executions that follow these rules.

More formally, an execution over a set of events AP is *well-defined* iff it satisfies all of the following formulas:

$$\Box(\mathit{start}(T, \mathsf{u}) \to \bigcirc\Diamond \mathit{end}(T, \mathsf{u}) \quad \forall \mathit{start}(T, \mathsf{u}), \mathit{end}(T, \mathsf{u}) \in AP \tag{4.1}$$

$$\neg\Diamond \mathit{start}(T, \mathsf{u}) \to \neg\Diamond \mathit{end}(T, \mathsf{u}) \quad \forall \mathit{start}(T, \mathsf{u}), \mathit{end}(T, \mathsf{u}) \in AP \tag{4.2}$$

$$\Box(p \to \bigcirc\Box\neg p) \quad \forall p \in AP \tag{4.3}$$

Specifically, Formula (4.1) captures that every start event must be followed by the corresponding end event, either in the next state or in a later one. Similarly, to ensure that every end event is also preceded by the corresponding start event, Formula (4.2) disallows end events to occur if the corresponding start event does not occur. Finally, Formula (4.3) requires that every event, be it start or end, can occur at most once.

4.3.5 Conformance and Consistency

We now have all the ingredients to formally define conformance of a schedule to a model and consistency of a model in terms of LTL. Let $\mathcal{M}$ be a process model and let s be a schedule. Then s *conforms* to $\mathcal{M}$ iff the execution corresponding to s is well-defined and satisfies the LTL formulas of all constraints in $\mathcal{M}$. More formally, we define e as the execution corresponding to s, ϕ as the conjunction of the formulas of all constraints in $\mathcal{M}$, and ψ as the conjunction of the formulas for well-definedness. Then s conforms to $\mathcal{M}$ iff $\phi \wedge \psi \vDash e$. Similarly, $\mathcal{M}$ is *consistent* iff there exists a schedule that conforms to $\mathcal{M}$, i.e. iff $\phi \wedge \psi$ is satisfiable.

Chapter 5

Checking Consistency

One advantage of having a process model in digital form and based on formal semantics is that a computer can perform automated reasoning on the model. Two common types of reasoning problems in logic are model checking and satisfiability checking. In our context, given a PRECISE process model, model checking corresponds to checking whether a particular schedule *conforms* to a given model, i.e. whether it satisfies all constraints in the model, whereas the problem of satisfiability asks whether the model is *consistent*, i.e. whether there is any schedule conforming to the model.

Since the formalism is based on LTL, and every process model can be translated into an LTL formula, the problem of checking consistency of a process model can be reduced to checking satisfiability of the corresponding LTL formula. There is a wide range of software tools able to solve LTL satisfiability problems we could use for this purpose. Checking satisfiability of LTL formulas, however, is PSPACE-complete (Sistla and Clarke, 1985). Thus, no efficient algorithm is known for checking satisfiability of large LTL formulas.

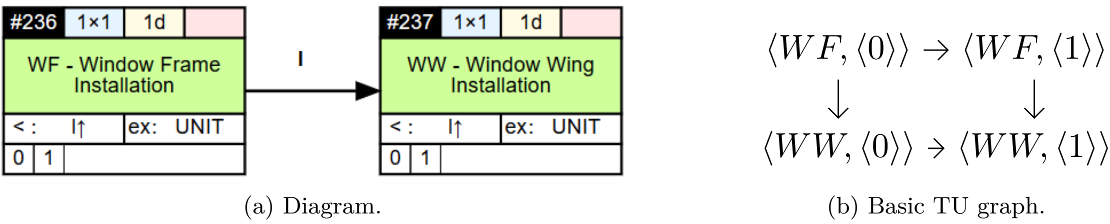
We performed some tests with the NuSMV model checker (Cimatti et al., 2002) using bounded model checking on the hotel example shown in Fig. 4.1. This took approximately 150 seconds on a standard desktop machine. If, however, the model is inconsistent, it takes much longer, and we aborted the test after hours. Therefore, we investigated other approaches for targeting the problem of checking consistency of PRECISE process models. The results are presented in this chapter. In particular, we first introduce a graph representation for process models and use properties on those graphs to characterize consistency of the corresponding process model. We start with a graph representation of simple models where only a subset of the kinds of constraints is allowed, and then extend the graph representation to accommodate more constraints until we consider all constraints. Then, we propose an algorithm that checks consistency of a model based on the introduced graph representation. Finally, we provide an upper bound for the computational complexity for the proposed algorithm, and present experiments showing that the algorithm performs well in practice.

5.1 Consistency of Basic Models

As a starting point, we investigate checking consistency of a simplified class of models where only tasks with orderings and basic precedences are allowed, whereas alternate precedence, chain precedence and exclusiveness constraints are forbidden. We call such models *basic* models. The main idea is to construct a directed graph where each node represents the execution of a task in a certain location, and arcs correspond to precedences between such executions. That is, we break down the diagram to the level of single CUs. We describe these graphs in more detail in the next section. Then, we will show that a basic process model is consistent iff the corresponding graph is cycle-free.

5.1.1 Basic Task-Unit Graph

Our hypothesis is that performing reasoning tasks on a basic process model, such as checking consistency, can be done more efficiently using a directed graph representation. This representation is called the *basic task-unit graph*, or *basic TU graph* for short, and corresponds to a break-down



(a) Diagram. (b) Basic TU graph.

Figure 5.1: Example of (a) a simple process model diagram and (b) the corresponding basic TU graph representation. The building is represented using a single attribute *level*, abbreviated as l.

of each task in the diagram into several nodes, one for each unit location of the task. Thus, nodes are pairs of activities and CUs.

Given a process model $\mathcal{M}$, the basic TU graph is a directed graph defined as $G_{\mathcal{M}} = \langle V, A \rangle$, where V is a set of nodes and $A \subseteq V \times V$ is a set of directed arcs between nodes. In particular, each node $v = \langle T, \mathsf{u} \rangle \in V$ represents the execution of an activity T in a CU u.

An arc $\langle v, v' \rangle \in A$ represents a precedence relationships from v to v' at the level of single nodes, and is conveniently written as $v \rightarrow v'$. In terms of events, each arc $\langle T, \mathsf{u} \rangle \rightarrow \langle T', \mathsf{u}' \rangle$ corresponds to the macro *end_to_start*$(T, \mathsf{u}, T', \mathsf{u}')$, i.e. it requires that ***end***(T', u') does not occur before ***start***(T, u) (See Table 4.1). Thus, arcs represent end-to-start precedences. In this way, we properly distinguish between start and end events. In particular, it is not necessary to explicitly represent the start event and end event of an activity in a CU in two different nodes.

The sets V and A are defined as follows. For a task t with activity T and CUs Cu, let $V(\mathsf{t}) := \{\langle T, \mathsf{u} \rangle \mid \mathsf{u} \in Cu\}$ denote the set of nodes of t. Then V is the union of all $V(\mathsf{t})$ for all tasks t. By design, if two tasks t, t' of the same activity T share a CU u, they both yield a node $\langle T, \mathsf{u} \rangle$, which is only contained once in V.

If there is an ordering constraint $\mathcal{O}$ on task t, we have an arc $v \rightarrow v' \in A$ for two nodes $v = \langle T, \mathsf{u} \rangle, v' = \langle T, \mathsf{u}' \rangle \in V(\mathsf{t})$ iff the ordering requires that T is executed first in u and only then in u', i.e. iff $\mathsf{u} <_{\mathcal{O},Cu} \mathsf{u}'$.

Finally, for a basic precedence from a task t to another task t' at scope s, we have an arc $v \rightarrow v' \in A$ for two nodes $v = \langle T, \mathsf{u} \rangle \in V(\mathsf{t})$, $v' = \langle T', \mathsf{u}' \rangle \in V(\mathsf{t}')$ iff u and u' are contained in the same construction area π at scope s, i.e. iff $\Pi_{\mathsf{s}}(\mathsf{u}) = \Pi_{\mathsf{s}}(\mathsf{u}')$.

Thus, orderings can be represented using arcs between nodes of the same task and different locations, while dependencies can be represented using arcs between nodes of different tasks.

As an example, consider a project of installing windows in a small house with the two levels 0 and 1. We want to capture that in each level, the frames of the windows must be installed before the wings, and both frames and wings must be installed in level 0 first and in level 1 second. Figure 5.1 shows (a) a diagram that captures these constraints as well as (b) the corresponding basic TU graph.

5.1.2 Cycles Correspond to Inconsistencies

In this section, we show that checking consistency of basic process models is equivalent to checking for cycles in the corresponding basic TU graph. We start with showing that cycles in the basic TU graph imply inconsistencies in the model.

Lemma 1. *A basic process model $\mathcal{M}$ is inconsistent if $G_{\mathcal{M}}$ has cycles.*

Proof. If a basic TU graph has a cycle involving two nodes $v = \langle T, \mathsf{u} \rangle$, $v' = \langle T', \mathsf{u}' \rangle$, this corresponds to a (transitive) precedence from v to v' and another precedence from v' to v. That is, activity T must be finished in u before activity T' can be started in u', and at the same time activity T' must be finished in u' before activity T can be started in u. Therefore, T can never start in u and T' can never start in u'. However, this violates the *existence* constraint, i.e. that for every task, the corresponding activity must be performed in all CUs of the task. Thus, there does not exist any schedule satisfying $\mathcal{M}$. □

Next, we show that the other direction holds as well. For that purpose, we define a *sequential* execution to be an execution where at most one activity is performed at a time, and in only one CU. Given a basic process model $\mathcal{M}$ and its basic TU graph $G_{\mathcal{M}} = \langle V, A \rangle$, a sequential execution of $\mathcal{M}$ can be written as a sequence $\langle v_1, \ldots, v_n \rangle$ of nodes $v_i \in V$ to denote an execution where each node is first started and then ended in the given sequence. Thus, one can obtain a sequence of states by replacing each node v_i with two consecutive states $\{\mathit{start}(v_i)\}$, $\{\mathit{end}(v_i)\}$. If all nodes $v_1, \ldots, v_n$ are pairwise different, the execution is well-defined. Now we show that a basic process model that is cycle-free is always consistent, because there is a well-defined sequential execution that conforms to it.

Lemma 2. *A basic process model $\mathcal{M}$ is consistent if $G_{\mathcal{M}}$ is cycle-free.*

Proof. If $G_{\mathcal{M}}$ is cycle-free, we can topologically sort its nodes to obtain a sequence of nodes $v_1, \ldots, v_n$, denoting a well-defined sequential execution of $\mathcal{M}$. By construction of $G_{\mathcal{M}}$, there is a node $\langle \mathsf{a}, \mathsf{u} \rangle$ for each activity that is required to be performed in CU u, so the execution constraints of the tasks are satisfied. Also, because the nodes are ordered topologically based on precedences in $G_{\mathcal{M}}$, the execution satisfies all precedences. Moreover, because the nodes are executed one after the other, we never execute two activities at the same time. Therefore, all implicit exclusiveness constraints is are satisfied. Thus, the obtained sequential execution satisfies the basic model $\mathcal{M}$, so $\mathcal{M}$ is consistent. □

Now that we have shown that both directions hold, we can conclude the following theorem.

Theorem 1. *A basic process model $\mathcal{M}$ is consistent iff $G_{\mathcal{M}}$ is cycle-free.*

Proof. By Lemma 1 and 2. □

Interestingly, there is no need to capture implicit exclusiveness constraints in the basic TU graph representation for checking consistency. Instead, it is sufficient to check whether precedences can be satisfied by checking for cycles in the basic TU graph. If all precedences can be satisfied, then all implicit exclusiveness constraints can be satisfied as well, because we can construct a sequential execution as shown in the proof of Lemma 2. From this we can extract the following corollary:

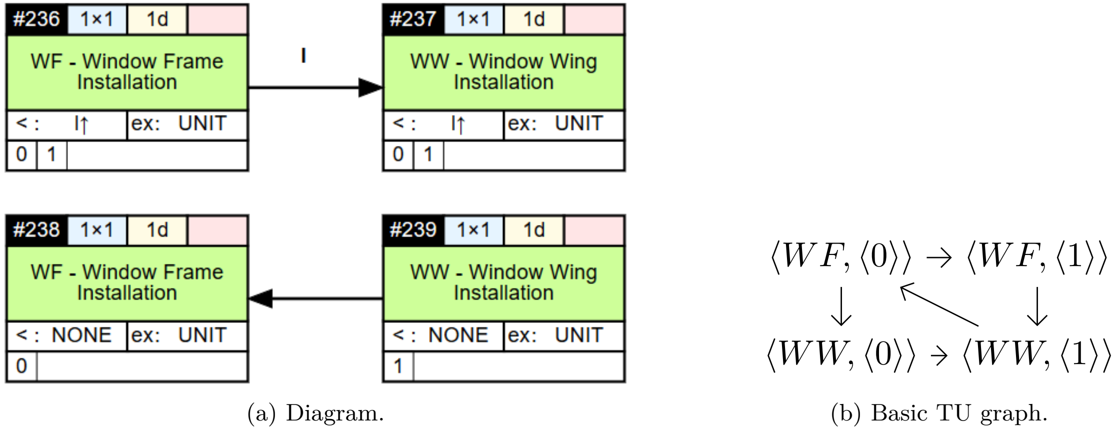
Corollary 1. *A basic process model $\mathcal{M}$ is consistent iff it has a conforming sequential execution.*

5.1.3 Cycles at the Diagram Level

One could ask whether checking for cycles at the diagram level would be enough to determine whether a model is consistent or not, i.e. whether constructing an auxiliary graph is necessary. In this section, we show that this is not possible in general, but works in restricted cases. In particular, it does not work if *i)* the diagram is cycle-free but the model is inconsistent, or *ii)* the diagram has a cycle but the model is consistent.

Inconsistent model with acyclic diagram Consider the process model $\mathcal{M}$ shown in Fig. 5.2(a). Note that the diagram is acyclic. If we ignore the two tasks at the bottom, the rest of the diagram is identical to the one shown in Fig. 5.1(a). Therefore, also the basic TU graph corresponding to the upper part of $\mathcal{M}$ would be the same as the one shown in Fig. 5.1(b), which is acyclic. The bottom part of $\mathcal{M}$, however, introduces another arc from $\langle WW, \langle 1 \rangle \rangle$ to $\langle WF, \langle 0 \rangle \rangle$. Therefore, the resulting basic TU graph, which is depicted in Fig. 5.2(b), is cyclic.

Consistent model with cyclic diagram Consider the process model $\mathcal{M}$ depicted in Fig. 5.3(a). Observe that Scaffolding Installation is to be performed in level -1, while Concrete Pouring is not. Therefore, the dependency between the two tasks does not apply to level -1, which allows to start Scaffolding Installation without waiting for anything else. Thus, while the diagram contains a cycle, the specific locations of the tasks and the scopes of the dependencies allow us to break the cycle and thus make the process model consistent. Accordingly, the basic TU graph corresponding to $\mathcal{M}$, which is shown in Fig. 5.3(b), is acyclic. Note that Fig. 5.3(c) shows an alternative diagram that yields the same basic TU graph, but which is acyclic.



(a) Diagram.

(b) Basic TU graph.

Figure 5.2: Example of (a) an acyclic diagram that translates to (b) a cyclic basic TU graph. The building is represented using a single attribute *level*, abbreviated as l.

On the one hand, these examples show that we cannot simply check for cycles in the diagram to understand whether there are cycles in the corresponding basic TU graph, and therefore whether the model is consistent.

On the other hand, they also suggest that if the diagram satisfies some restrictions, then there is a correspondence between acyclic diagrams and consistent models. This restriction requires that two tasks of the same activity must not have *overlapping* locations. Two tasks of the same activity have overlapping locations iff there is at least one CU shared location by both tasks.

The following two theorems define this correspondence between consistent models and acyclic diagrams with non-overlapping tasks more formally.

Theorem 2. *If a basic process model $\mathcal{M}$ has an acyclic diagram and non-overlapping locations, then $\mathcal{M}$ is consistent.*

Proof. Assume $\mathcal{M}$ is inconsistent. Then, by Theorem 1, $G_{\mathcal{M}}$ has a cycle. Because locations are non-overlapping, for every node $v = \langle T, \mathsf{u} \rangle$, there is exactly one task that has activity T and location u, which is the one that caused the creation of the node v. The cycle in $G_{\mathcal{M}}$ is thus lifted to a cycle in the diagram of $\mathcal{M}$. But this leads to a contradiction because we assume $\mathcal{M}$ is acyclic. □

Theorem 3. *If $\mathcal{M}$ is a basic process model, with a diagram that possibly contains cycles, and $G_{\mathcal{M}}$ is cycle-free, then there is an acyclic diagram with non-overlapping locations corresponding to the same basic TU graph $G_{\mathcal{M}}$.*

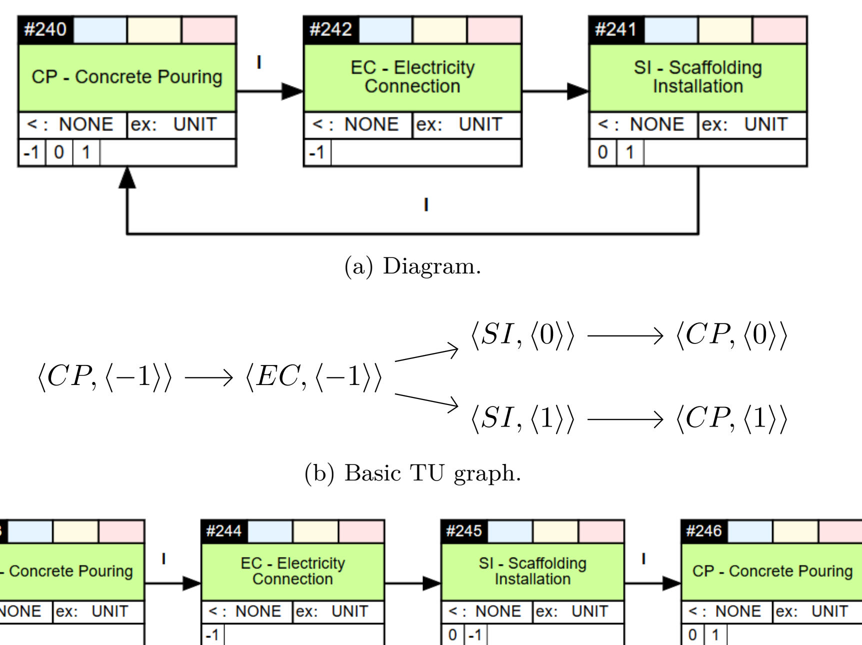
Proof. We show how to construct an acyclic diagram with non-overlapping locations from $G_{\mathcal{M}} = \langle V, A \rangle$. In particular, for each task $\langle T, \mathsf{u} \rangle$ we create a task of activity T and with a single CU u. Similarly, for each arc $v \to v' \in A$ we add a basic precedence at global scope from the task corresponding to v to the task corresponding to v'. Thus, we create a diagram that is structurally identical with $G_{\mathcal{M}}$. Because $G_{\mathcal{M}}$ is acyclic, the created diagram is also acyclic. And because every task contains a single location, and every pair of activity and CU cannot appear more than once in V, locations in the diagram are non-overlapping. Finally, it is easy to see that the resulting diagram translates back to $G_{\mathcal{M}}$. □

Given the results of Theorems 2 and 3, it is reasonable to disallow cycles and overlapping locations at the diagram level. An acyclic diagram with non-overlapping locations is called *well-structured*. By Theorem 2, every basic process model with a well-structured diagram is consistent. Theorem 3 implies that every basic models can be defined using a well-structured diagram.

For more general models, however, these properties do not hold. Therefore, in the following discussion, we do not assume diagrams to be well-structured unless otherwise noted.

5.2 Consistency of Exclusive Models

In Section 5.1, we showed how to check consistency of a basic process model by means of the basic TU graph. We also showed that if a process model has a well-structured diagram, the model is



(a) Diagram.

(b) Basic TU graph.

(c) Equivalent diagram.

Figure 5.3: Example of (a) a cyclic process model diagram that translates to (b) an acyclic basic TU graph, as well as (c) an alternative diagram that translates to the same basic TU graph, but which is acyclic. The building is represented using a single attribute *level*, abbreviated as l.

consistent. In this section, we consider *exclusive* models, where all the constraints of basic models plus exclusiveness are allowed, while alternate precedence and chain precedence are forbidden. We start showing that also exclusive models are always consistent if they have a well-structured diagram. Next, we introduce an extension of basic TU graphs that captures exclusiveness constraint and show how it relates to consistency of exclusive process models.

5.2.1 Assuming Well-Structured Diagrams

We show that every exclusive process model with a well-structured diagram is consistent:

Theorem 4. *If $\mathcal{M}$ is an exclusive process model with a well-structured diagram, then $\mathcal{M}$ is consistent.*

Proof. We show that $\mathcal{M}$ is consistent by showing that there exists a sequential execution that satisfies all the constraints. To construct such an execution, we first topologically order all the tasks in the diagram, which is possible because the diagram is well-structured. Then the tasks are performed in the obtained sequence, meaning that a task in the sequence does not start until the previous task has been performed in all locations where it is required. For performing a task in a set of locations we consider the order given by the ordering constraints. If no ordering constraint exists, any order among locations can be used. The resulting sequence conforms to a model that is stricter than $\mathcal{M}$, where all dependencies and exclusiveness constraints have global scope. By construction, ordering constraints are also satisfied. Therefore, the sequence conforms to $\mathcal{M}$. $\square$

If an exclusive process model is not well-structured, it still may be consistent. Consider Fig. 5.4 for an example. It shows an exclusive process model $\mathcal{M}$ with a cyclic diagram. Consider the following sequence of nodes:

$$\langle A, \langle 0, c\rangle\rangle, \langle B, \langle 1, c\rangle\rangle, \langle B, \langle 1, r\rangle\rangle, \langle D, \langle 1, k\rangle\rangle,$$

$$\langle A, \langle 0, k\rangle\rangle, \langle C, \langle 0, r\rangle\rangle, \langle C, \langle 0, b\rangle\rangle, \langle D, \langle 1, b\rangle\rangle$$

Note that all exclusiveness constraints of $\mathcal{M}$ are satisfied. For example, the two nodes $\langle A, \langle 0, c\rangle\rangle$ and $\langle A, \langle 0, k\rangle\rangle$, which must have exclusive access to level 0, are only interrupted by tasks in level 1,

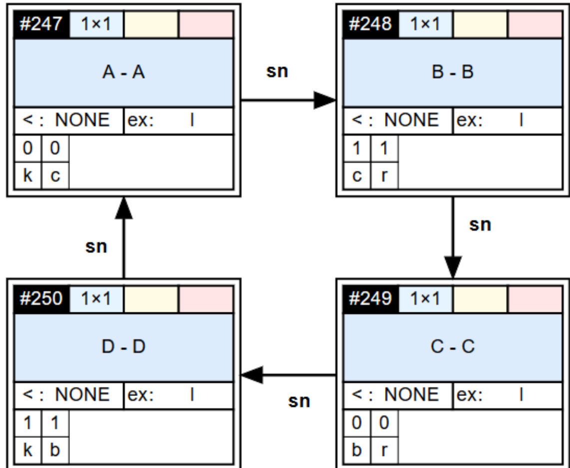


Figure 5.4: A consistent exclusive process model with a cyclic diagram for which no equivalent acyclic diagram exists, because exclusiveness constraints cannot be split. Locations are represented by attributes *level* with a range of 0 and 1, and *section* (*sn*) with a range of kitchen (k), corridor (c), room (r), and bathroom (b).

which does not violate the exclusiveness constraint. Dependencies are satisfied as well, and indeed the given sequence of nodes is a sequential execution conforming to $\mathcal{M}$.

As opposed to basic models, where we could split tasks to make a process model acyclic, this is not possible here, because splitting the tasks would also split the CUs to which the exclusiveness constraints apply. Therefore, we introduce an extension of basic TU graphs that is able to capture exclusiveness constraints. This is useful to characterize consistency of exclusive process models at a lower level without depending on well-structured diagrams.

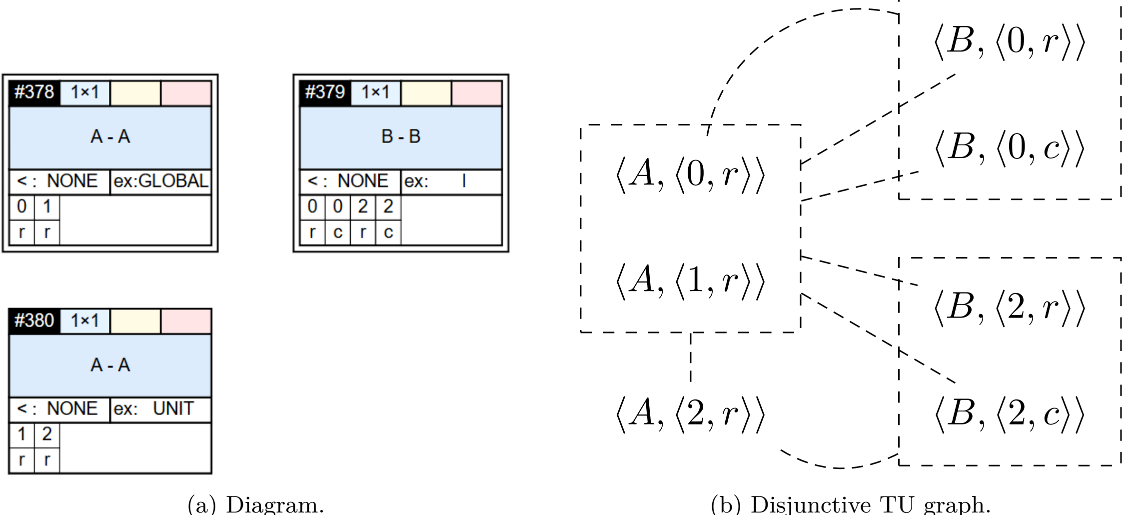
5.2.2 Disjunctive TU Graphs

The basic TU graph representation introduced in Section 5.1 only captures precedence constraints, i.e. orderings and basic dependencies. Now we extend this representation by a notion that captures exclusiveness constraints. For that purpose, let us recap the semantics of exclusiveness constraints.

Consider an exclusive process model $\mathcal{M}$ with an exclusiveness constraint at scope $\mathtt{s}$ on task $\mathtt{t}_1$ with locations Cu. This requires that while task $\mathtt{t}_1$ has started but not finished its execution in a CA π of scope $\mathtt{s}$, it has exclusive access to π, i.e. no other task $\mathtt{t}_2$ can be executed in π during that time. From the perspective of $\mathtt{t}_2$, this means that its execution in a location $\mathtt{u}_2$ must occur either before or after all CUs $\mathtt{u}_1 \in Cu$ such that $\Pi_{\mathtt{s}}(\mathtt{u}_1) = \Pi_{\mathtt{s}}(\mathtt{u}_2)$.

This semantics is defined in terms of the *uninterrupted* macro. In particular, we use the macro to relate a set of nodes x of $\mathtt{t}_1$ that are contained in a CA at scope $\mathtt{s}$ to single nodes v of other tasks $\mathtt{t}_2$, requiring that v occurs either before all nodes in x or after all of them. Therefore, to capture this constraint in a graph representation, we extend the basic TU graph by an additional set of undirected edges, where each edge $e = \langle x, v \rangle$ connects a set of nodes x to a single node v. The set of nodes x is called an *exclusive group*, the undirected edge $e = \langle x, v \rangle$ is called a *disjunctive edge*, and the whole graph representation is called *disjunctive TU graph*. The disjunctive TU graph representation is inspired from disjunctive graphs. A disjunctive graph is a mixed graph with both directed arcs and undirected edges, and is useful to represent jobshop scheduling problems, which are a special class of resource constrained scheduling problems. Nodes correspond to operations, directed arcs correspond to precedences among operations, and undirected edges connect nodes that require the same non-shareable resource. For more information on jobshop problems and the disjunctive graph representation, see e.g. (Fortemps and Hapke, 1997).

Before we define the disjunctive TU graph in more detail, let us look at an example. Consider Fig. 5.5, which shows (a) an exclusive process model $\mathcal{M}$ and (b) the corresponding disjunctive graph $G_{\mathcal{M}}$, where dashed boxes indicate exclusive groups, containing the corresponding nodes, and dashed lines represent disjunctive edges. The model contains three tasks $\mathtt{t}_A, \mathtt{t}'_A, \mathtt{t}_B$, where $\mathtt{t}_A$ and $\mathtt{t}'_A$ are two tasks of the same activity A with overlapping locations, whereas task $\mathtt{t}_B$ is of



(a) Diagram.

(b) Disjunctive TU graph.

Figure 5.5: Example of (a) an exclusive process model and (b) the corresponding disjunctive TU graph. The building is represented by two attributes *level* and *section*, abbreviated as l and sn. Exclusive groups are displayed as dashed boxes, and disjunctive edges are displayed as dashed lines.

activity B. All tasks belong to the same phase, which structures the building using the attributes *level* and *section*. Task $\mathtt{t}_A$ has a global exclusiveness constraint and task $\mathtt{t}_B$ has an exclusiveness constraint of scope *level*.

To capture the global exclusiveness constraint on task $\mathtt{t}_A$, there is an exclusive group x_A containing all nodes of $\mathtt{t}_A$, and a disjunctive edge from x_A to every other node in the graph. In particular, considering the nodes $v_1 = \langle A, \langle 1, r \rangle \rangle$, $v_2 = \langle A, \langle 2, r \rangle \rangle$ of task $\mathtt{t}$, there is a disjunctive edge from x_A to v_2 but not to v_1, because v_1 is already contained in x_A whereas v_2 is not. Similarly, to capture the exclusiveness constraint at scope *level* on task $\mathtt{t}_B$, the nodes of $\mathtt{t}_B$ are partitioned by level into two exclusive groups $x_{B,0}$, $x_{B,2}$, where $x_{B,0}$ contains nodes of level 0 and $x_{B,2}$ contains nodes of level 2. Each of the two exclusive groups is connected to the other nodes $G_\mathcal{M}$ that share the same level. Specifically, $x_{B,0}$ is connected to $\langle A, \langle 0, r \rangle \rangle$, and $x_{B,2}$ is connected to $\langle A, \langle 2, r \rangle \rangle$.

We now define the disjunctive TU graph in more detail. For a given exclusive process model $\mathcal{M}$, the disjunctive TU graph of $\mathcal{M}$ is defined as $G_\mathcal{M} = \langle V, A, X, E \rangle$, where V is a set of nodes, A is a set of conjunctive arcs, X is a set of *exclusive groups* and E is a set of *disjunctive edges*. The nodes V and the arcs A are defined as in the basic TU graph and correspond to *existence* constraints and *end_to_start* constraints, respectively. The set of exclusive groups $X \subseteq 2^V$ contains sets of nodes that can be connected to single nodes by disjunctive edges. Accordingly, the set of disjunctive edges $E \subseteq X \times V$ is a relation from exclusive groups to single nodes.

To show how X and E are constructed, we allow the projection Π_s to be applied to a node $\langle \mathsf{a}, \mathsf{u} \rangle$ by defining

$$\Pi_\mathsf{s}(\langle \mathsf{a}, \mathsf{u} \rangle) = \Pi_\mathsf{s}(\mathsf{u})$$

The projection of a set of nodes $V' \in V$ is defined in the same way as the projection of a set of CUs, i.e. it is the set of the projections of each element in the set:

$$\Pi_\mathsf{s}(V') = \{\Pi_\mathsf{s}(v') \mid v' \in V'\}$$

Similarly, the selection of the nodes in a given set of nodes $V' \in V$ based on a CA π_s of scope $\mathtt{s}$ is defined as the set of those nodes whose projection to $\mathtt{s}$ equals π_s, i.e.:

$$\sigma_{\pi_\mathsf{s}}(V') = \{v' \mid \Pi_\mathsf{s} = v', v' \in V'\}$$

Now we can construct the sets X and E as follows: For each task $\mathtt{t}$ with an exclusiveness constraint at scope $\mathtt{s}$, and for each CA $\pi_\mathsf{s} \in \Pi_\mathsf{s}(V(\mathtt{t}))$, let $x = \sigma_{\pi_\mathsf{s}}(V(t))$ be the exclusive group of the nodes of $\mathtt{t}$ contained in π_s. We add x to X, and for each other node $v \in \sigma_{\pi_\mathsf{s}}(V) \setminus x$ that is also contained in the same CA, we add a disjunctive edge $\langle x, v \rangle$ to E. Similarly to the corresponding

LTL formula, we must not add a disjunctive edge between an exclusive group x and and a node v if v is contained in x, because that would mean that v must occur before itself, which is a contradiction and differs from the semantics of the corresponding formula. This is important to correctly handle two tasks t, t' of the same activity with overlapping locations, because then also the sets of nodes $V(\mathsf{t})$, $V(\mathsf{t}')$ of the two tasks are overlapping.

5.2.3 Orientations of Disjunctive TU Graphs

In Section 5.1, we showed that a basic process model is inconsistent iff the corresponding basic TU graph has cycles. In Section 5.2.2, we introduced the disjunctive TU graph representation to capture the constraints in exclusive models. What is missing is a characterization of consistency of exclusive models in terms of properties of the disjunctive graph representation. In this section, we will close this gap. In particular, we will give a characterization in terms of *orientations* of a disjunctive TU graph. Therefore, we first define the concept of orientation.

Consider a disjunctive TU graph $G_{\mathcal{M}} = \langle V, A, X, E\rangle$ with a disjunctive edge $e = \langle x, v\rangle \in E$. The *orientation* of e is a graph $G'_{\mathcal{M}} = \langle V, A', X, E'\rangle$ where e is either replaced with the set of arcs $\{v\} \times x$ from v to all nodes in x or with the same arcs in the other direction, i.e. with $x \times \{v\}$. We use the following notation to conveniently refer to set of arcs resulting resolving a disjunctive edge $\langle x, v\rangle$ in a particular direction:

$$x \to v := x \times \{v\}$$
$$v \to x := \{v\} \times x$$

An *orientation* of the disjunctive TU graph $G_{\mathcal{M}}$ is a basic TU graph $\vec{G}_{\mathcal{M}} = \langle V, A''\rangle$ that is the result of orienting each disjunctive edge in E. Further, $G_{\mathcal{M}}$ is *orientable* iff there is an acyclic orientation of $G_{\mathcal{M}}$.

The idea is that if an execution conforms to an exclusive process model $\mathcal{M}$, then for every disjunctive edge between two sets of nodes, one set of nodes finishes before the other starts. Thus, every execution corresponds to one orientation of the corresponding disjunctive TU graph. Moreover, we know from Section 5.1 that a basic TU graph G' can only be satisfied by a schedule if G' is acyclic, so we derive the following theorem:

Theorem 5. *Let $\mathcal{M}$ be an exclusive process model and let $G_{\mathcal{M}} = \langle V, A, X, E\rangle$ be the corresponding disjunctive TU graph. Then $\mathcal{M}$ is consistent iff $G_{\mathcal{M}}$ is orientable.*

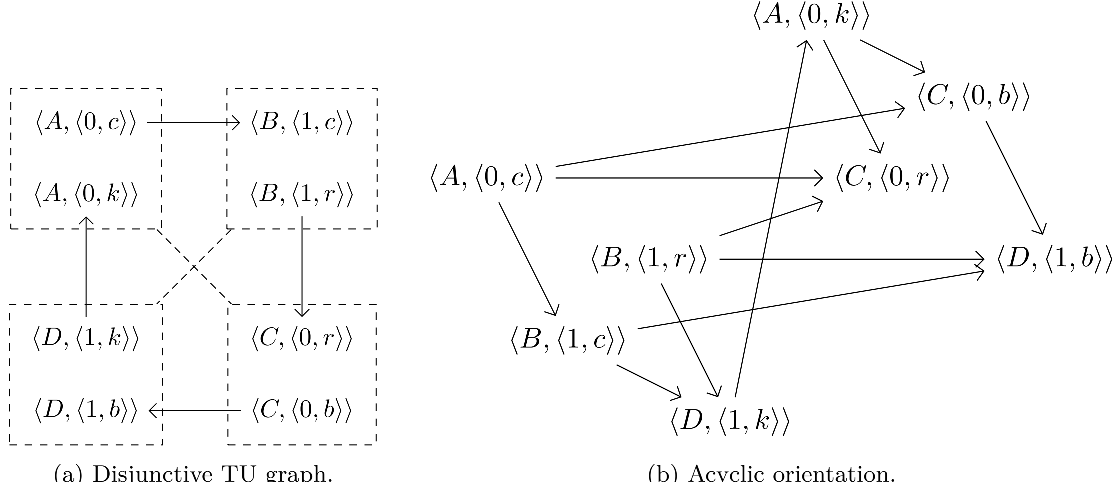
Proof. Assume $G_{\mathcal{M}}$ is orientable. Then it has an acyclic orientation $\vec{G}_{\mathcal{M}} = \langle V, A'\rangle$ of $G_{\mathcal{M}}$. Then we can topologically sort $\vec{G}_{\mathcal{M}}$ to obtain a well-defined sequential execution. Because $A \subseteq A'$, all precedence constraints are satisfied. Also, by construction of $\vec{G}_{\mathcal{M}}$, for every disjunctive edge $\langle x, v\rangle$ in E, v occurs either before or after all nodes in x. Then, by construction of $G_{\mathcal{M}}$, all exclusiveness constraints are satisfied. Hence, the sequential execution conforms to $\mathcal{M}$, so $\mathcal{M}$ is consistent.

Next, suppose $G_{\mathcal{M}}$ is not orientable. Then, there is no acyclic orientation of $G_{\mathcal{M}}$, i.e. every orientation of $G_{\mathcal{M}}$ is cyclic. Then, there is no execution that corresponds to any orientation of $G_{\mathcal{M}}$, thus there is no execution satisfying $\mathcal{M}$, so $\mathcal{M}$ is inconsistent. □

As an example, consider Fig. 5.6, which shows (a) the disjunctive TU graph $G_{\mathcal{M}}$ of the process model $\mathcal{M}$ shown in Fig. 5.4 as well as (b) an acyclic orientation $\vec{G}_{\mathcal{M}}$ of $G_{\mathcal{M}}$. According to Theorem 5, this implies that the corresponding model, i.e. the one shown in Fig. 5.4, is consistent. Indeed, in Section 5.2, we provide a sequential execution that conforms to the model. In particular, that execution is a topological order of $\vec{G}_{\mathcal{M}}$.

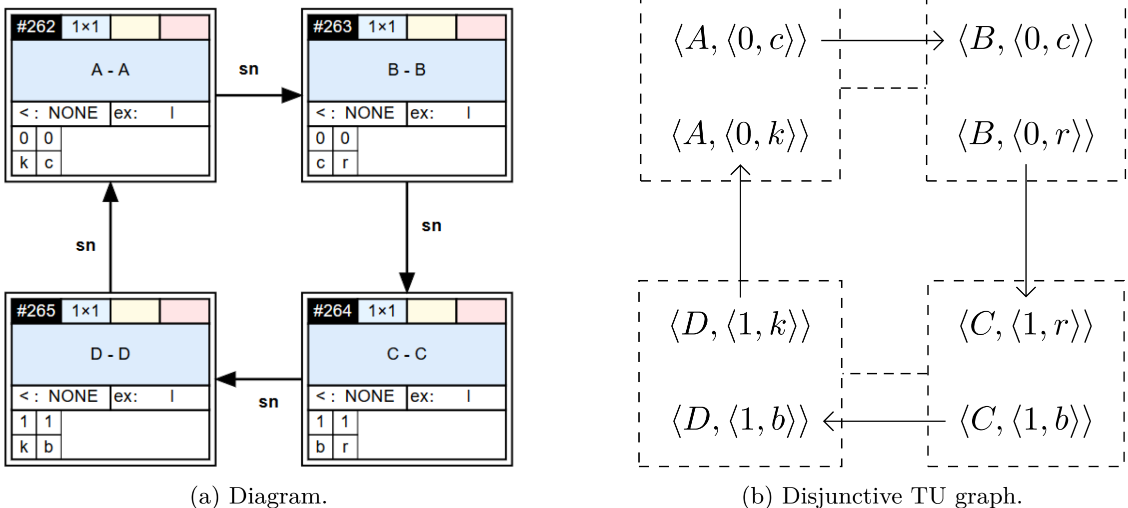
Note that for convenience, in Fig. 5.6(a) and in following figures, we replace many disjunctive edges with a single disjunctive edge between two exclusive groups. Specifically, a disjunctive edge between two exclusive groups denotes all disjunctive edges from one group to all nodes of the other group and vice-versa. In the example, the disjunctive edge between the exclusive group x_A of nodes of activity A to the exclusive group x_C of nodes of activity C, for instance, denotes a disjunctive edge between x_A and a node v_C for all nodes $v_C \in x_C$, as well as all disjunctive edge between x_C and a node v_A for all nodes $v_A \in x_A$. We use the same convention for all following figures.

Also, observe that Corollary 1 still holds for exclusive models, because we for every consistent model, we can construct a sequential execution. This is also the reason why we do not need to capture implicit exclusiveness constraints in the disjunctive TU graph representation.



(a) Disjunctive TU graph. (b) Acyclic orientation.

Figure 5.6: The (a) disjunctive TU graph $G_{\mathcal{M}}$ of the process model $\mathcal{M}$ shown in Fig. 5.4 and (b) an acyclic orientation of $G_{\mathcal{M}}$ that results from executing A before C and B before D.



(a) Diagram. (b) Disjunctive TU graph.

Figure 5.7: Example of (a) an inconsistent exclusive process model and (b) the corresponding disjunctive TU graph. The disjunctive TU graph is acyclic in its conjunctive arcs but has no acyclic orientation.

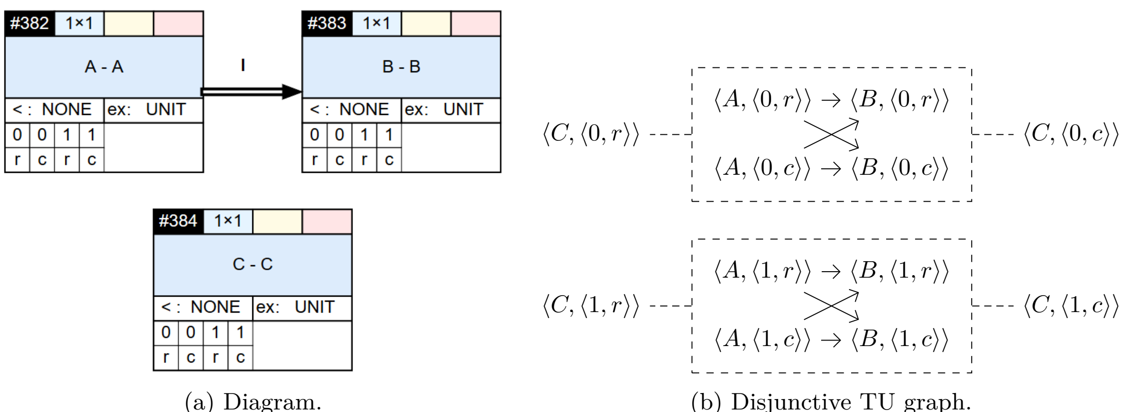
Note that for a given exclusive process model, it is possible that a disjunctive TU graph is acyclic in its conjunctive arcs but no acyclic orientation exists. An example of such an exclusive process model and the corresponding disjunctive TU graph is depicted in Fig. 5.7.

5.3 Consistency of General Models

In this section, we discuss how to check consistency of general models, i.e. with all constraints allowed. In particular, we now also allow alternate precedence and chain precedence. In contrast to basic and exclusive models, a general model can be inconsistent even if the diagram is well-structured. Therefore, we focus on our graph representation for checking these models.

In Section 5.2.2 we introduced the disjunctive TU graph representation of exclusive process models and showed that a model is consistent iff the corresponding disjunctive TU graph is orientable. In this section, we show how to construct disjunctive TU graphs that also capture alternate precedences and chain precedences.

First, recall that for all dependencies, the semantics of basic precedence always applies. Alternate precedences and chain precedences only add further constraints to the constraints imposed by



(a) Diagram.

(b) Disjunctive TU graph.

Figure 5.8: Example of (a) a process model with a chain precedence and (b) the corresponding disjunctive TU graph.

the corresponding basic precedence. Thus, the set of nodes and the set of conjunctive arcs can be defined as shown in Section 5.1. Furthermore, note that the constraints added by alternate precedences and chain precedences are defined in terms of the LTL macro *uninterrupted* and thus can be captured by disjunctive edges. Intuitively, both chain precedences and alternate precedences as well as exclusiveness constraint share a common concept of blocking the execution of certain nodes while another set of nodes is being executed. The only difference among these three types of constraints is how the sets of nodes that block each other are selected.

5.3.1 Chain Precedences

Apart from the basic precedence constraints, a chain precedence between two tasks can be seen as an exclusiveness constraint that spans over the two tasks. To capture this in a disjunctive TU graph $G_{\mathcal{M}} = \langle V, A, X, E\rangle$, we add exclusive groups and disjunctive edges to X and E, respectively:

For each chain precedence at scope $\mathtt{s}$ from $\mathtt{t}$ to $\mathtt{t}'$, and for each CA $\pi_{\mathtt{s}}$ of scope $\mathtt{s}$ in which at least one of the two tasks is to be performed, i.e. for each

$$\pi_{\mathtt{s}} \in \Pi_{\mathtt{s}}(V(\mathtt{t})) \cup \Pi_{\mathtt{s}}(V(\mathtt{t}'))$$

let x be the exclusive group of the nodes of $\mathtt{t}$ and $\mathtt{t}'$ contained in $\pi_{\mathtt{s}}$, defined as:

$$x = \sigma_{\pi_{\mathtt{s}}}(V(\mathtt{t}) \cup V(\mathtt{t}'))$$

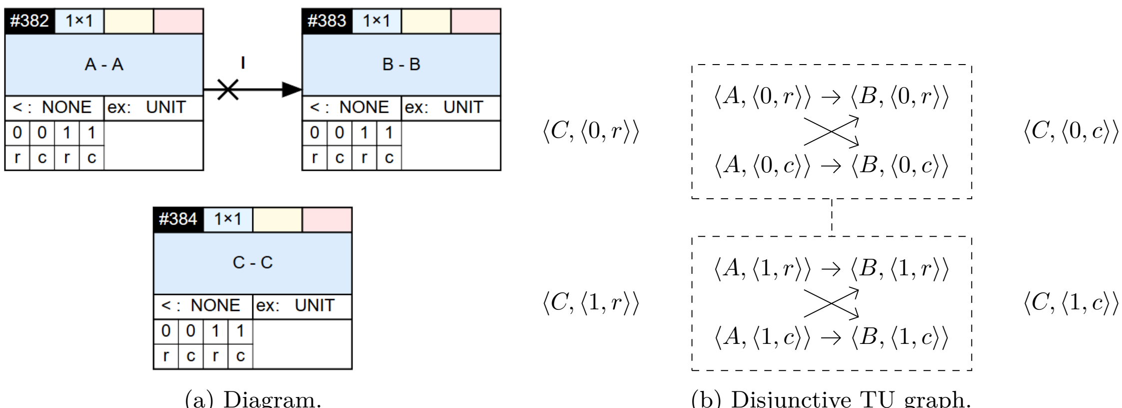
We add x to X, and for each other node $v \in \sigma_{\pi_{\mathtt{s}}}(V) \setminus x$ that is also contained in the same CA, we add a disjunctive edge $\langle x, v\rangle$ to E.

For an example, consider the process model given in Fig. 5.8(a). The corresponding TU graph is shown in Fig. 5.8(b). In the model, there are three tasks A, B, C and a chain precedence from A to B at scope *level*. In the disjunctive TU graph, the nodes of tasks A and B are partitioned into two exclusive groups $x_{A,B,0}$, $x_{A,B,1}$, where $x_{A,B,0}$ contains all nodes with a CU in level 0, and $x_{A,B,1}$ contains all nodes with a CU in level 1.

5.3.2 Alternate Precedences

Similar to chain precedences, alternate precedences also add a constraint on top of a basic precedence. For an alternate precedence at scope $\mathtt{s}$ between two tasks $\mathtt{t}$ and $\mathtt{t}'$, this additional constraint requires that for each two CAs at scope $\mathtt{s}$ shared among $\mathtt{t}$ and $\mathtt{t}'$, both tasks must be finished in all CUs of one CA before they can start in the other CA. To capture this in the disjunctive TU graph, we add an exclusive group for each shared CA of the two tasks and connect each pair of exclusive groups by a disjunctive edge.

Thus, we now extend the disjunctive TU graph representation $G_{\mathcal{M}} = \langle V, A, X, E\rangle$ by defining the set of disjunctive edges $E \subseteq X \times X$ as a relation between two exclusive groups rather than a relation between an exclusive group and a single node. For still being able to connect an exclusive



(a) Diagram.

(b) Disjunctive TU graph.

Figure 5.9: Example of (a) a process model with an alternate precedence and (b) the corresponding disjunctive TU graph.

group x to a single node v as required by exclusive constraint and chain precedences, i.e. to upgrade the previous representation to the one now introduced, we add an exclusive group $\{v\}$ to X and replace the disjunctive edge $\langle x, v\rangle$ with $\langle x, \{v\}\rangle$. Accordingly, the orientation of a disjunctive edge $e = \langle x, x'\rangle$ is now defined as the replacement of e with either the set of arcs from all nodes in x to all nodes in x', denoted as $x \to x'$, or the same arcs in the other direction, denoted as $x' \to x$.

We now show how to use this new representation to capture alternate precedences in more detail. Specifically, for a disjunctive TU graph $\langle V, A, X, E\rangle$, we add exclusive groups and disjunctive edges to X and E as follows:

For each alternate precedence at scope $\mathtt{s}$ from $\mathtt{t}$ to $\mathtt{t}'$, and for each two CAs $\pi_\mathtt{s}$, $\pi'_\mathtt{s}$ of scope $\mathtt{s}$ where both $\mathtt{t}$ and $\mathtt{t}'$ are to be performed, i.e. for each

$$\pi_\mathtt{s}, \pi'_\mathtt{s} \in \Pi_\mathtt{s}(V(\mathtt{t})) \cap \Pi_\mathtt{s}(V(\mathtt{t}')), \pi_\mathtt{s} \neq \pi'_\mathtt{s}$$

let x, x' be the exclusive groups of the nodes of $\mathtt{t}$ and $\mathtt{t}'$ contained in $\pi_\mathtt{s}$, $\pi_\mathtt{s}$, respectively, defined as:

$$x = \sigma_{\pi_\mathtt{s}}(V(\mathtt{t}) \cap V(\mathtt{t}'))$$
$$x' = \sigma_{\pi'_\mathtt{s}}(V(\mathtt{t}) \cap V(\mathtt{t}'))$$

We add x and x' to X, and we add a disjunctive edge $\langle x, x'\rangle$ to E.

For an example, consider the process model $\mathcal{M}$ given in Fig. 5.9(a). The corresponding TU graph is shown in Fig. 5.9(b). The model is identical to the process model $\mathcal{M}'$ shown in Fig. 5.8(a), except that the chain precedence is changed to an alternate precedence.

Observe that the disjunctive TU graph $G_\mathcal{M}$ resulting from $\mathcal{M}$ is similar to the disjunctive TU graph $G_{\mathcal{M}'}$ resulting from $\mathcal{M}'$, which is shown in Fig. 5.8(b). Both $G_\mathcal{M}$ and $G_{\mathcal{M}'}$ contain the same nodes, the same arcs, and the same exclusive groups. Only the exclusions of exclusive groups and therefore the disjunctive edges differ. While in $G_{\mathcal{M}'}$, for each exclusive group of the chain precedence from A to B, there is a disjunctive edge to each node of other tasks (i.e. C), in $G_\mathcal{M}$ we only have a disjunctive edge among two exclusive groups coming from the same alternate precedence. And because there is only one alternate precedence with only two exclusive groups, we only have a single disjunctive edge between those two exclusive groups.

5.3.3 A Uniform Representation

We finally defined how to construct a disjunctive TU graph that captures all constraints of a given process model $\mathcal{M}$, where $\mathcal{M}$ possibly contains all types of constraints. Because we are able to capture all constraints without extending the disjunctive TU graph representation introduced in Section 5.2.2, it is not surprising that also for a general process model $\mathcal{M}$ it holds that $\mathcal{M}$ is consistent iff the corresponding disjunctive TU graph $G_\mathcal{M}$ is orientable.

Theorem 6. *Let $\mathcal{M}$ be a (general) process model. Then $\mathcal{M}$ is consistent iff the corresponding disjunctive TU graph $G_\mathcal{M}$ is orientable.*

Proof. By construction of $G_{\mathcal{M}}$ and Theorem 5. □

Also, as in the case of basic models and exclusive models, every consistent model has a sequential execution, i.e. Corollary 1 holds for general models as well:

Corollary 2. *A process model $\mathcal{M}$ is consistent iff it has a conforming sequential execution.*

Finally, note that in a disjunctive TU graph $G_{\mathcal{M}} = \langle V, A, X, E\rangle$, the nodes V correspond to the LTL *existence* constraints, the arcs A correspond to the LTL macro *end_to_start*, and the disjunctive edges E correspond to the LTL macro *uninterrupted*. Therefore, we can extend the modeling language by new types of constraints without needing to change or extend the disjunctive TU graph representation, as long as the new constraints are defined in terms of the same three base constraints.

For example, suppose we need a new constraint that requires that two tasks $\mathtt{t} = \langle T, Cu\rangle$, $\mathtt{t}' = \langle T', Cu'\rangle$ must be executed one after the other in CAs at a given scope $\mathtt{s}$. This can be thought of as an exclusiveness constraint at scope $\mathtt{s}$ on the two tasks that only applies between them, whereas other tasks are not affected by this constraint. We can define this in terms of the macro *uninterrupted* as follows:

$$\forall \pi_{\mathtt{s}} \in \Pi_{\mathtt{s}}(Cu) \cap \Pi_{\mathtt{s}}(Cu')$$
$$\mathit{uninterrupted}(\{\langle T, \sigma_{\pi_{\mathtt{s}}}(Cu)\rangle\}, \{\langle T', \sigma_{\pi_{\mathtt{s}}}(Cu')\rangle\}$$

This can be captured in the disjunctive TU graph by defining the corresponding exclusive groups and exclusions.

5.4 Checking Orientability

In Section 5.3 and particularly in Theorem 6 we showed that a given process model is consistent iff the corresponding disjunctive TU graph is orientable, i.e. iff it has an acyclic orientation. In this section, we discuss how to check whether a disjunctive TU graph has an acyclic orientation.

Brute-force One option would be a brute-force approach, i.e. generating all possible orientations and checking whether some of them is acyclic. To understand whether such an approach is practical, let us analyze the computational complexity.

Consider a process model $\mathcal{M}$ and the corresponding disjunctive TU graph $G_{\mathcal{M}} = \langle V, A, X, E\rangle$. An orientation of $G_{\mathcal{M}}$ is the result of orienting each disjunctive edge $e \in E$ in one of the two possible directions. So there are $2^{|E|}$ orientations of $G_{\mathcal{M}}$. Each of these orientations is a basic TU graph $\vec{G} = \langle V, \vec{A}\rangle$, where $\vec{A}$ is a superset of A containing also the arcs with which the disjunctive edges where replaced. Because, disjunctive edges connect sets of nodes, the size of A is in $\mathcal{O}(|A| + |V|^2)$. Finally, one can check in $\mathcal{O}(|V| + |\vec{A}|)$ time whether $\vec{G}$ is cyclic (Tarjan, 1972). In total, we get that the time needed for checking all orientations of $G_{\mathcal{M}}$ is in $\mathcal{O}(2^{|E|} \cdot (|V|^2 + |A|))$.

To express this in terms of the model, let n be the number of tasks and m the number of CUs. The number of disjunctive edges depends on exclusiveness constraints, alternate precedences and chain precedences. Each exclusiveness constraint on a task partitions introduces an exclusive group for each CA at the given scope, and possibly a disjunctive edge from each other node in the graph to one exclusive group. The number of nodes is in $\mathcal{O}(nm)$, and because there can be an exclusiveness constraint on each tasks, the total number of disjunctive edges introduced by exclusiveness constraints is in $\mathcal{O}(mn^2)$. Similarly, also chain precedences between two tasks connect each node of other tasks to one exclusive group. The number of chain precedences in a model is in $\mathcal{O}(n^2)$, so the number of disjunctive edges introduced by all of them is in $\mathcal{O}(n^3m)$. The number of alternate precedences is the same as that of chain precedences. Yet, the number of disjunctive edges introduced by a single one is different. Specifically, the nodes of the two tasks are partitioned into CAs to obtain the exclusive groups, and the exclusive groups are pairwise connected by a disjunctive edge. Because there is at most one exclusive group per CU, the total number of disjunctive edges introduced by alternate precedences is in $\mathcal{O}(n^2m^2)$. So in total, the number of disjunctive edges in $G_{\mathcal{M}}$ is in $\mathcal{O}(n^3m + n^2m^2)$. Thus, to check whether a process model is consistent using this brute force approach takes $\mathcal{O}(2^{n^3m+n^2m^2} \cdot (n^4m^4))$. Hence, the time grows exponentially with the number of disjunctive edges, which grows polynomially with the number of tasks and locations in $\mathcal{M}$. This shows that the brute-force approach is not practical, so we need to come up with better approaches.

5.4.1 A Simple Recursive Algorithm

Recall that the orientation of a disjunctive TU graph is defined as the result of orienting each disjunctive edge in the graph. In this section, we investigate how the orientation of a single edge relates to the existence of an acyclic orientation to derive an initial algorithm that is superior to the brute-force approach.

First, consider a disjunctive TU graph $G_{\mathcal{M}} = \langle V, A, X, E \rangle$. Suppose $G_{\mathcal{M}}$ has already a cycle in A, i.e. the corresponding basic TU graph $\langle V, A \rangle$ is cyclic. Then also $G_{\mathcal{M}}$ is said to be *cyclic*, and it cannot be orientable.

Lemma 3. *If a disjunctive TU graph $G_{\mathcal{M}} = \langle V, A, X, E \rangle$ is cyclic, then it is not orientable.*

Proof. Every orientation of $G_{\mathcal{M}}$ contains all arcs of A. Thus, if A contains a cycle, then that cycle is preserved in all orientations of $G_{\mathcal{M}}$. □

Second, recall that an orientation of a disjunctive TU graph $G_{\mathcal{M}}$ is defined by orienting all disjunctive edges in one of the two possible directions. Also, orientating a single disjunctive edge $\langle x_1, x_2 \rangle$ in both directions $x_1 \to x_2$ and $x_2 \to x_1$ of a $G_{\mathcal{M}}$ results in two more disjunctive TU graph $G_{\mathcal{M}_{1,2}}$ and $G_{\mathcal{M}_{2,1}}$, respectively. This implies that the set of acyclic orientations of $G_{\mathcal{M}}$ is equal to the union of the acyclic orientations of B $G_{\mathcal{M}_{1,2}}$ and $G_{\mathcal{M}_{2,1}}$:

Lemma 4. *Let $G_{\mathcal{M}} = \langle V, A, X, E \rangle$ be a disjunctive TU graph, and let $e = \langle x_1, x_2 \rangle \in E$ be a disjunctive edge of $G_{\mathcal{M}}$. Further, let $G_{\mathcal{M}_{1,2}}$ and $G_{\mathcal{M}_{2,1}}$ be the disjunctive graphs obtained by orienting e in the direction $x_1 \to x_2$ and $x_2 \to x_1$, respectively. Then a basic TU graph $\vec{G} = \langle V, A' \rangle$ is an acyclic orientation of $G_{\mathcal{M}}$ iff it is an acyclic orientation of $G_{\mathcal{M}_{1,2}}$ or of $G_{\mathcal{M}_{2,1}}$.*

Proof. Assume $\vec{G}$ is an acyclic orientation of $G_{\mathcal{M}}$. By definition of orientation, $\vec{G}$ must have an orientation of e in one of the two directions. If the direction is $x_1 \to x_2$, then $\vec{G}_{\mathcal{M}}$ is also an acyclic orientation of $G_{\mathcal{M}_{1,2}}$. Otherwise the direction is $x_2 \to x_1$, and $\vec{G}_{\mathcal{M}}$ is also an acyclic orientation of $G_{\mathcal{M}_{2,1}}$.

Next, assume $G'_{\mathcal{M}}$ is an acyclic orientation of either $G_{\mathcal{M}_{1,2}}$ or $G_{\mathcal{M}_{2,1}}$. Then for all disjunctive edges in $E \setminus \{e\}$, $\vec{G}_{\mathcal{M}}$ has orientations in some direction. By construction of $G_{\mathcal{M}_{1,2}}$ and $G_{\mathcal{M}_{2,1}}$, $\vec{G}_{\mathcal{M}}$ has also an orientation of e. Hence, $\vec{G}_{\mathcal{M}}$ is an acyclic orientation of $G_{\mathcal{M}}$. □

Thus, to find an acyclic orientation of a disjunctive TU graph $G_{\mathcal{M}}$, we can choose some disjunctive edge, compute the orientations in both directions, and recursively search an acyclic orientation in the resulting disjunctive TU graph. In particular, there are two base cases: *i)* a cycle is found, so by Lemma 3 there is no acyclic orientation, and *ii)* the graph is acyclic and has no disjunctive edges, so it is already an acyclic orientation of itself. If none of these base cases applies, we choose a random edge, orient it in both directions, and recursively apply the algorithm on the result. By Lemma 4, if one of the two recursive calls returns an acyclic orientation, we can return it. Otherwise, there is no acyclic orientation.

The single steps are shown in more detail in Algorithm 1. The algorithm is implemented by the function `findAcyclicOrientation`, which returns a pair of a boolean and a disjunctive TU graph. The boolean indicates whether the returned graph represents an acyclic orientation of $G_{\mathcal{M}}$ or not. If not, the graph represents the node of the recursion tree where the problem was identified.

Algorithm 1 has a much better best-case performance than the brute-force approach, because it detects cycles earlier. In particular, if the input disjunctive TU graph $G_{\mathcal{M}} = \langle V, A, X, E \rangle$ is already cyclic, this can be detected in $\mathcal{O}(|V| + |A|)$ (Tarjan, 1972). The general time complexity, however, is the same as the one of the brute-force approach, because we end up with generating all possible schedules and checking them for cycles. Therefore, in the following sections, we propose several ideas for improving Algorithm 1 by simplifying the graph to be checked.

5.4.2 Ignoring Simple Disjunctive Edges

A first observation is that those disjunctive edges where both sides consist of one single node never introduce cycles. We call such disjunctive edges *simple* disjunctive edges. Thus, we can ignore these disjunctive edges and thereby speedup the algorithm.

Algorithm 1 Find an acyclic orientation of a disjunctive TU graph.

```
1: function findAcyclicOrientation(G_M = ⟨V, A, X, E⟩)
2:     if G_M has a cycle then
3:         return ⟨FALSE, G_M⟩
4:     else if E is empty then
5:         return ⟨TRUE, G_M⟩
6:     else
7:         Choose a random disjunctive edge ⟨x, x′⟩ ∈ E
8:         return tryBothDirections(G_M, x, x′)
9:     end if
10: end function

11: function tryBothDirections(G_M = ⟨V, A, X, E⟩, x, x′)
12:     ⟨success, G′⟩ ← tryDirection(G_M, x, x′)
13:     if success then
14:         return ⟨TRUE, G′⟩
15:     end if
16:     ⟨success, G′⟩ ← tryDirection(G_M, x′, x)
17:     if success then
18:         return ⟨TRUE, G′⟩
19:     end if
20:     return ⟨G_M, FALSE⟩
21: end function

22: function tryDirection(G_M = ⟨V, A, X, E⟩, x_s, x_t)
23:     Copy G_M into G′_M
24:     orient(G′_M, x_s, x_t)
25:     return findAcyclicOrientation(G′_M)
26: end function

27: procedure orient(G_M = ⟨V, A, X, E⟩, x_s, x_t)
28:     Remove ⟨x_s, x_t⟩ from E
29:     for all v_s ∈ x_s, v_t ∈ x_t do
30:         Add ⟨v_s, v_t⟩ to A
31:     end for
32: end procedure
```

Lemma 5. *Let $G_{\mathcal{M}} = \langle V, A, X, E\rangle$ be a disjunctive TU graph, and let $G'_{\mathcal{M}} = \langle V, A, X, E \setminus E_1\rangle$ be the result of removing all simple edges E_1 from E. Then $G_{\mathcal{M}}$ is orientable iff $G'_{\mathcal{M}}$ is orientable.*

Proof. Clearly, if $G_{\mathcal{M}}$ is orientable, then so is $G'_{\mathcal{M}}$. If $G'_{\mathcal{M}}$ is orientable, then by Corollary 2, there is a sequential execution that satisfies all constraints in $G'_{\mathcal{M}}$. Because it is sequential, the execution also satisfies all simple disjunctive edges in E_1. Then the sequential execution satisfies all constraints in $G_{\mathcal{M}}$, and hence $G_{\mathcal{M}}$ is orientable. □

5.4.3 Resolving Disjunctive Edges

One problem with the approach of Algorithm 1 is that the disjunctive edges and the directions in which to orient them are chosen blindly. Instead, we should first orient those disjunctive edges where only an orientation in one direction is possible because the other direction would immediately lead to a cycle. For a given disjunctive edge between two sets of nodes, this is the case iff there is a path from a node in one set to a node in the other set. In order to not introduce a cycle, this edge must be *resolved* to the single direction that follows that path. We show this more formally in the following lemma:

Lemma 6. *Let $G_{\mathcal{M}} = \langle V, A, X, E\rangle$ be a disjunctive TU graph, and let $e = \langle x_1, x_2\rangle \in E$ be a disjunctive edge. If there is a path $v_1 \to \cdots \to v_2$ from a node $v_1 \in x_1$ to a node $v_2 \in x_2$, then $G_{\mathcal{M}}$ is orientable iff the disjunctive TU graph $G_{\mathcal{M}_{1,2}}$ obtained by orienting e in the direction $x_1 \to x_2$ is orientable.*

Proof. Let $G_{\mathcal{M}_{2,1}}$ be the disjunctive TU graph obtained by orienting the disjunctive edge e in $G_{\mathcal{M}}$ in the other direction $x_2 \to x_1$. Then $G_{\mathcal{M}_{2,1}}$ has a cycle $v_1 \to \cdots \to v_2 \to v_1$. By Lemma 3, $G_{\mathcal{M}_{2,1}}$ is not orientable. Then, by Lemma 4, a basic TU graph G' is an acyclic orientation of $G_{\mathcal{M}}$ it is also an acyclic orientation of $G_{\mathcal{M}_{1,2}}$. Then $G_{\mathcal{M}}$ is orientable iff $G_{\mathcal{M}_{1,2}}$ is orientable. □

We can use Lemma 6 to devise an algorithm that simplifies a given disjunctive TU graph by resolving as many disjunctive edges as possible. Algorithm 2 shows this procedure in detail. It iterates through all disjunctive edges and resolves them if possible. Because resolving disjunctive edges introduces new directed arcs, it is possible that a disjunctive edge that could not be resolved in the first run can be resolved later. Therefore, we repeatedly iterate through disjunctive edges until no more edges can be resolved. Also, if a cycle is detected, there is no point resolving edges any further because we already know by Lemma 3 that there is no acyclic orientation, so we stop.

Algorithm 2 Resolve edges in a disjunctive TU graph.

```
 1: procedure resolveEdges(G_M = ⟨V, A, X, E⟩)
 2:     fixpoint ← FALSE
 3:     while G_M is acyclic ∧ ¬fixpoint do
 4:         fixpoint ← TRUE
 5:         for all ⟨x1, x2⟩ ∈ E do
 6:             r ← tryResolving(G_M, x1, x2) ∨
                        tryResolving(G_M, x2, x1)
 7:             fixpoint ← fixpoint ∧ ¬r
 8:         end for
 9:     end while
10: end procedure

11: function tryResolving(G_M = ⟨V, A, X, E⟩, xs, xt)
12:     if some vs ∈ xs reaches some vt ∈ xt over A then
13:         orient(G_M, xs, xt)
14:         return TRUE
15:     else
16:         return FALSE
17:     end if
18: end function
```

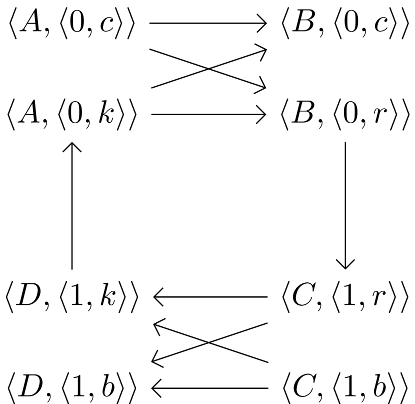


Figure 5.10: Result of resolving disjunctive edges in the disjunctive TU graph shown in Fig. 5.7(b).

As an example, reconsider the process model $\mathcal{M}$ and the corresponding disjunctive TU graph $G_{\mathcal{M}} = \langle V, A, X, E \rangle$ shown in Fig. 5.7. We already observed that $G_{\mathcal{M}}$ has no cycles, but is not orientable. Now we will use Lemma 6, and in particular Algorithm 2, to show why this is the case.

In $G_{\mathcal{M}}$, for each task there is an exclusive group containing all the nodes of the task. We use x_A, x_B, x_C, x_D to refer to the exclusive group of the task of activity A, B, C, D, respectively. Figure 5.7(b) displays two disjunctive edges $\langle x_A, x_B \rangle$, $\langle x_C, x_D \rangle$ between exclusive groups, where each of them denotes a disjunctive edge from one exclusive group to each node of the other group and vice-versa.

Consider x_A and x_B. There is an arc from a node in x_A to a node in x_B. Regardless of whether we consider x_A and x_B to be directly connected by a single disjunctive edge $\langle x_A, x_B \rangle$, or whether we consider all combinations of disjunctive edges between one exclusive group and all nodes of the other exclusive group, by Lemma 6, all nodes of x_A must come before all nodes of x_B. Therefore, we resolve the disjunctive edge $\langle x_A, x_B \rangle$ in the direction $x_A \rightarrow x_B$, i.e. we replace it with the corresponding arcs. Similarly, there is an arc from a node in x_C to a node in x_D, so we resolve $\langle x_C, x_D \rangle$ in the direction $x_C \rightarrow x_D$.

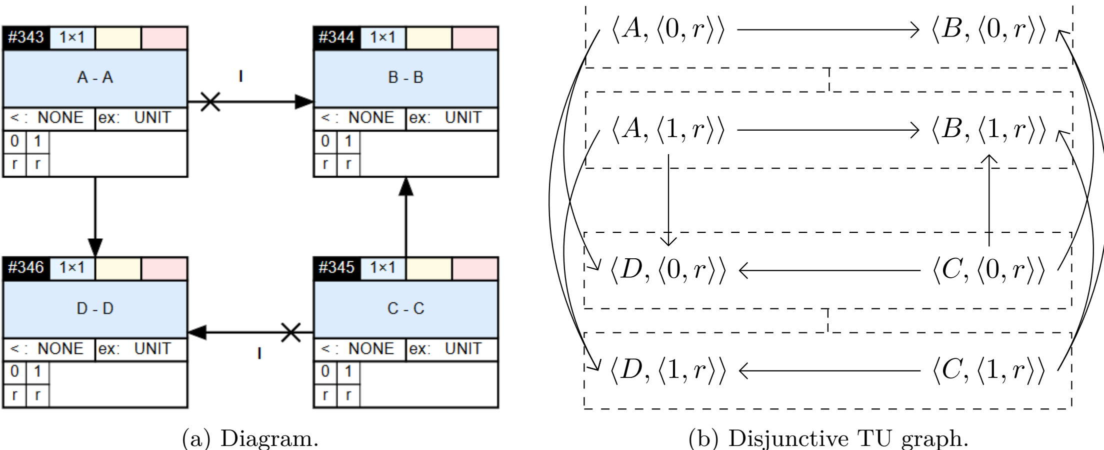
The result of resolving disjunctive edges in $G_{\mathcal{M}}$ is shown in Fig. 5.10. Note that there is a cycle, so $G_{\mathcal{M}}$ is not orientable. Thus, by resolving disjunctive edges, we revealed a problem in the model .

There are, however, process models that are inconsistent even if the corresponding disjunctive TU graphs are not cyclic, and neither does resolving disjunctive edges according to Lemma 6 lead to cycles. Specifically, this is the case if a disjunctive TU graph has multiple disjunctive edges which, if viewed separately, can be oriented in both directions without introducing cycles, but when all disjunctive edges are oriented, there is always a cycle. Such disjunctive edges that cannot be satisfied together because they influence each other are called *deadlocks*.

An example of a process model $\mathcal{M}$ and the corresponding disjunctive TU graph $G_{\mathcal{M}}$ that contains deadlocks is given in Fig. 5.11. In the beginning, we can start executing activity A in any CU, as well as activity C in any CU. But then we are stuck. Because of the alternate precedences, we can execute neither A nor C in another CU before B or D are completed in the previous CUs where we already started A or C, respectively. And because of the basic precedences at global scope from A to D and from C to B, we cannot start B nor D in any CU before A or C are completed in all CUs.

If we look at the disjunctive TU graph $G_{\mathcal{M}}$, we observe that there is no cycle. Also, it is not possible to resolve any disjunctive edge, because for each disjunctive edge between two exclusive groups, there is no path from one exclusive group to the other. Yet, once we choose any direction for one disjunctive edge, the other one cannot be satisfied, because if we also orient that one in some direction, a cycle is introduced. Hence, $G_{\mathcal{M}}$ is not orientable.

Note that we came to this conclusion by choosing a random orientation for each disjunctive edge, which follows the idea of Algorithm 1. Thus, resolving edges using Algorithm 2 cannot replace Algorithm 1, because otherwise we would not be able to correctly check consistency of process



(a) Diagram. (b) Disjunctive TU graph.

Figure 5.11: Example of (a) a process model with a deadlock, i.e. where (b) the corresponding disjunctive TU graph is not orientable even if resolving disjunctive edges does not lead to cycles. The inconsistency is only detected if both directions for one disjunctive edge are tried out.

models with deadlocks. Instead, to take advantage of resolving, we can follow the main steps of Algorithm 1, but resolve as many disjunctve edges as possible before choosing a random direction for a random edge.

5.4.4 Partitioning Into Independent Subgraphs

Another opportunity for improving the algorithm is by partitioning a disjunctive TU graph $G_\mathcal{M}$ into multiple subgraphs $G_1, \ldots, G_n$ such that all arcs and disjunctive edges between different subgraphs can be easily satisfied. Then we know that problems can only occur within subgraphs, so we only have to check whether each subgraph G_i is orientable to determine whether also $G_\mathcal{M}$ is orientable. Thus, in contrast to resolving edges, where we look for disjunctive edges that can be oriented in at most one direction without introducing cycles, we know look for edges that can be oriented in at least one direction.

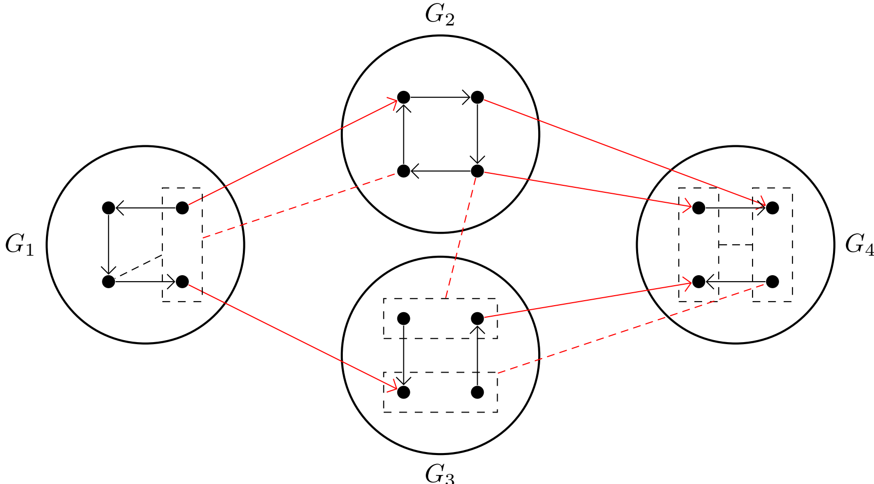
First, we define the partition of a disjunctive TU graph. A *partition* of a disjunctive TU graph $G_\mathcal{M} = \langle V, A, X, E\rangle$ is a set P of disjoint and non-empty node subsets $S \subseteq V$ called *parts*, such that for each node $v \in V$, there is exactly one part $S \in P$ with $v \in S$. A part $S \in P$ *induces* a subgraph $G_\mathcal{M}[S] := \langle S, A_S, X_S, E_S\rangle$ that contains only those arcs and disjunctive edges between nodes contained in S, i.e. where

$$A_S = A \cap (S \times S)$$
$$X_S = X \cap 2^S$$
$$E_S = E \cap X_S \times X_S$$

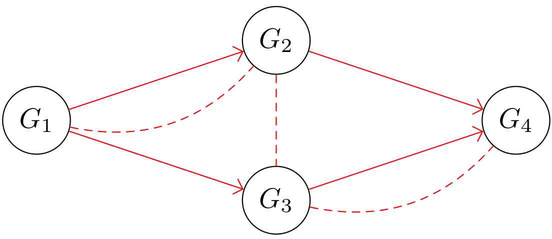
The arcs, exclusive groups, and disjunctive edges of $G_\mathcal{M}$ that are not contained in any subgraph induced by parts of P, because they have nodes of different subgraphs, are called *cut arcs*, *cut groups*, and *cut edges*, respectively.

As an example, consider the disjunctive TU graph partition P shown in Fig. 5.12(a), where G_1, G_2, G_3, G_4 are the induced subgraphs. For the moment, we can ignore the inside of each subgraph. Instead, we focus on cut arcs, cut groups, and cut edges (displayed in red). For this purpose, it is sufficient to consider a more abstract disjunctive TU graph G_P, called the *collapsed* graph of a partition P, where each subgraph is collapsed to a single node. The collapsed graph G_P resulting from P is shown in Fig. 5.12(b).

Observe that G_P is acyclic. This means that there is a topological order on the subgraphs, e.g. $\langle G_1, G_2, G_3, G_4\rangle$. All cut arcs are satisfied if we execute the subgraphs in such an order. Also, note that there are no cut groups, i.e. every exclusive group is fully contained in one subgraph and is collapsed into a single node in G_P. Thus, all disjunctive edges in G_P connect two single nodes.



(a) Disjunctive TU graph partition.



(b) Subgraphs collapsed to single nodes.

Figure 5.12: Example of (a) a partition of disjunctive TU graph $G_{\mathcal{M}}$ partitioned into subgraphs G_1, G_2, G_3, G_4. Cut arcs and cut edges are shown in red. This partition is proper and acyclic, because collapsing each subgraph into a single node, as shown in (b), yields an disjunctive TU graph that is acyclic and where disjunctive edges connect two single nodes.

Therefore, these disjunctive edges as well as the corresponding cut edges in $G_{\mathcal{M}}$ are also satisfied by following a topological order on the subgraphs.

Conversely, if there were any cut groups involved in disjunctive edges, it might not be possible to satisfy them by following a topological order on subgraphs. As an example, consider Fig. 5.13. It is possible to topologically order the subgraphs, but executing them in the obtained order violates the cut edge.

Hence, there are two desirable properties of partitions. A partition P is called *acyclic* iff the corresponding collapsed graph G_P is acyclic, and is called *proper* iff there are no cut groups. If a partition is both proper and acyclic, we can apply the following lemma:

Lemma 7. *Let $G_{\mathcal{M}} = \langle V, A, X, E\rangle$ be a disjunctive TU graph, and let $P = \{S_1, \ldots, S_n\}$ be a proper and acyclic partition of $G_{\mathcal{M}}$. Then $G_{\mathcal{M}}$ is orientable iff for all parts $S_i \in P$, $G_{\mathcal{M}}[S_i]$ is orientable.*

Proof. First, suppose there is a part $S_i \in P$ such that $G_{\mathcal{M}}[S_i]$ is not orientable. Note that $G_{\mathcal{M}}[S_i]$ is a subgraph of $G_{\mathcal{M}}$, i.e. all nodes, arcs, and disjunctive edges of $G_{\mathcal{M}}[S_i]$ are contained in $G_{\mathcal{M}}$. Then $G_{\mathcal{M}}$ is not orientable either.

Next, suppose that for each $S_i \in P$, $G_{\mathcal{M}}[S_i]$ is orientable. Then $G_{\mathcal{M}}[S_i]$ has an acyclic orientation $G_i = \langle S_i, A_i\rangle$. We show how to construct an acyclic orientation of $G_{\mathcal{M}}$. First, note that $G_{\mathcal{M}}$ is itself acyclic, because P is acyclic and all G_i are acyclic, and each arc in A is either a cut arc or contained in some subgraph G_i. Therefore, we must orient all disjunctive edges in E without introducing cycles. For non-cut edges, such orientations are already contained in the corresponding sugraphs. Therefore, we can orient these edges in $G_{\mathcal{M}}$ by removing them from E and instead add all arcs from all subgraphs G_i. Because P is proper, each exclusive groups is fully contained in a single subgraph. Thus, we can find an orientation of all cut edges by topologically ordering subgraphs according to cut arcs, i.e. by topologically ordering the nodes of the collapsed subgraph

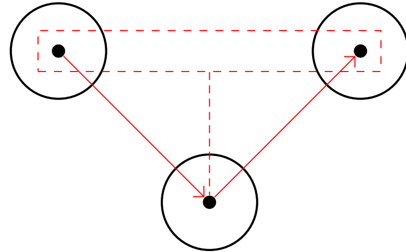


Figure 5.13: Example of an acyclic partition of a disjunctive TU graph with a cut edge between a cut group (displayed in red) and another subgraph that cannot be oriented in any direction without introducing cycles.

G_P. Each cut edge $e \in E$ is then oriented in the direction that follows the obtained topological order on the corresponding subgraphs. In this way, the topological order is always respected and no cycle is introduced. The resulting graph is an acyclic orientation of the original $G_{\mathcal{M}}$. Hence, $G_{\mathcal{M}}$ is orientable. □

Thus, to improve the overall algorithm, we now focus on devising an algorithm that finds a proper and acyclic partition of a given disjunctive TU graph $G_{\mathcal{M}} = \langle V, A, X, E \rangle$. The idea is to build a directed graph such that its Strongly Connected Components (SCCS) form a proper and acyclic partition of $G_{\mathcal{M}}$. In contrast to disjunctive TU graphs, not all the arcs of this auxiliary graph represent precedence constraints. Instead, we add more arcs between nodes that possible interfere such that we are sure that each SCC can be checked separately.

As a starting point, consider the basic TU graph $G'_{\mathcal{M}} = \langle V, A \rangle$ obtained by simply ignoring all disjunctive edges. Then $G_{\mathcal{M}}$ and $G'_{\mathcal{M}}$ have the same SCCs. By definition of SCCs, there is always a topological order on SCCs, because otherwise there is a cycle involving multiple SCCs, but then they are strongly connected and therefure should be contained in one single SCC. Thus, a partition P of $G_{\mathcal{M}}$ into SCCs is acyclic. Yet, P is not necessarily proper. To make P proper, we must guarantee that every exclusive groups is fully contained in a single SCC of G'. Thus, for each exclusive group $x \in X$, we ensure that every node $v \in x$ can reach every other node $v' \in x$. We can achieve this by adding arcs in both directions between such nodes v, v'. More precisely, we define the set C of *clustering arcs* as

$$C := \{v \to v' \mid v, v' \in x, x \in X, v \neq v'\}$$

The *cluster graph* of $G_{\mathcal{M}}$ is defined as $G^c_{\mathcal{M}} = \langle V, A \cup C \rangle$, and the *clusters* of $G_{\mathcal{M}}$ are the SCCs of $G^c_{\mathcal{M}}$. We now show that the clusters of a disjunctive TU graph form a proper and acyclic partition.

Lemma 8. *Let $G_{\mathcal{M}} = \langle V, A, X, E \rangle$ be a disjunctive TU graph, and let P be a partition of $G_{\mathcal{M}}$ into its clusters. Then P is proper and acyclic.*

Proof. Let $G^c_{\mathcal{M}} = \langle V, A \cup C \rangle$ be the cluster graph of $G_{\mathcal{M}}$. By definition, P consists of the SCCs of $G^c_{\mathcal{M}}$. First, observe that each side of a disjunctive edge $e \in E$ forms a complete subgraph in $G^c_{\mathcal{M}}$. Hence, each side of e is strongly connected and thus fully contained in one SCC. Therefore, P is proper. Next, observe that among SCCs of a directed graph, there is never a cycle, because otherwise the SCCs involved in that cycle would all be strongly connected and therefore be all contained in one single SCC. Because also $G^c_{\mathcal{M}}$ contains all arcs of $G_{\mathcal{M}}$, P is acyclic. □

Note that the partiton shown in Fig. 5.12 is a partition into clusters. That is, within each subgraph, every node can reach every other node either over arcs or by being contained in the same side of a disjunctive edge. Each of these subgraphs represents an area of potential problems. Applying Algorithm 1, and optionally Algorithm 2, to each subgraph reveals that G_3 is orientable whereas G_1, G_2 and G_4 are not. Specifically, G_2 contains a cycle, and G_1 as well as G_4 contain a disjunctive edge that cannot be oriented in any direction without introducing cycles.

The advantage of first partitioning the whole graph into smaller subgraphs is to reduce the complexity by narrowing down the areas in which problems can potentially occur. These problems within subgraphs can be found using the algorithms proposed previously. The number of subgraphs in which a disjunctive TU graph $G_{\mathcal{M}}$ can be partitioned depends on the structure $G_{\mathcal{M}}$. In

particular, certain disjunctive TU graphs only have a single cluster and thus cannot be partitioned into smaller subgraphs. For instance, after partitioning a disjunctive TU graph $G_{\mathcal{M}}$ into clusters, yielding subgraphs $G_1, \ldots, \mathcal{G}_n$, it is not possible to further partition any of the resulting subgraphs, because all of them have a single cluster.

In practice, however, if diagrams are well-structured, most tasks only have simple exclusiveness constraints, and most dependencies are basic precedences, then partitioning the corresponding disjunctive TU graph into clusters will often lead to many small subgraphs, which can be checked for consistency without taking too much time.

As a final remark, note that partitioning a disjunctive TU graph not only allows to restrict the checks to smaller subgraphs, but also allows to run the checks in parallel. In a multi-core environment, this improves scalability with the number of available CPU cores.

5.4.5 Putting It All Together

In Section 5.4.1, we proposed a simple but inefficient algorithm that is able to find an acyclic orientation of a disjunctive TU graph. Then we proposed several ideas for for simplifying a given disjunctive TU graph. In this section, we propose an improved algorithm that is a combination of the ideas introduced so far.

Consider a disjunctive TU graph $G_{\mathcal{M}} = \langle V, A, X, E \rangle$ that is to be checked for orentability. First, by Lemma 5, any simple disjunctive edges in E can be removed. Next, we partition $G_{\mathcal{M}}$ into its clusters. By Lemma 8 and 7, we know that $G_{\mathcal{M}}$ is orientable iff each subgraph $G_{\mathcal{M}}[S]$ induced by a cluster S is orientable. Thus, it is enough to check each subgraph separately. To do so, we first apply Lemma 6 and resolve as many disjunctive edges as possible in each subgraph. Now we do not have any further means to simplify the resulting subgraphs. Therefore, for each subgraph, we proceed along the lines of Algorithm 1. If there is a cycle, the subgraph is not orientable. Otherwise, if no disjunctive edges are left, the graph is already an acyclic orientation. Otherwise, we choose some random edge and try out the two possible orientations, applying this algorithm recursively, starting by further partitioning the resolved subgraphs into clusters.

Algorithm 3 shows these steps in more detail. It is implemented by the function `check`, which returns a tuple of the form $\langle \mathit{success}, G_{\mathcal{M}}, R \rangle$, where *success* is a boolean indicating whether the check was successful, $G_{\mathcal{M}}$ is the input graph, and R is a list of results of the same structure corresponding to the clusters into which $G_{\mathcal{M}}$ was partitioned. In this way, the result is a tree corresponding to the recursive partitioning of $G_{\mathcal{M}}$, where each node in the tree indicates whether all nodes in the subtree below it are successful. This is to avoid the need of merging the partitions of a graph into an actual orientation at every level of the recursion if not needed.

If we only need to know whether $G_{\mathcal{M}}$ is orientable, we only have to check whether the root result node is successful without the need of building an actual orientation. If instead an orientation is needed, e.g. to support scheduling, and $G_{\mathcal{M}}$ is indeed orientable, we can build an acyclic orientation from the result tree as shown in the proof of Lemma 8. To facilitate this, we use Tarjan's SCCs algorithm (Tarjan, 1972) for computing the SCCs of the cluster graph of $G_{\mathcal{M}}$, because it automatically sorts SCCs topologically. The details of building an actual orientation in this way are shown in Section 6.2.

5.4.6 Complexity

We now analyze the general complexity of Algorithm 3. For that purpose, we construct a worst-case scenario. Assume that there is a non-orientable disjunctive TU graph $G_{\mathcal{M}} = \langle V, A, X, E \rangle$ where partitioning never applies and where no disjunctive edge can be resolved. Then Algorithm 3 performs the same recursive steps as Algorithm 1. On every recursive step with a partially oriented disjunctive TU graph $G'_{\mathcal{M}}$, which has a number of arcs in $\mathcal{O}(|A|)$, the algorithm tries to partition $G'_{\mathcal{M}}$ into clusters and to resolve edges in $G'_{\mathcal{M}}$, but to no avail.

Partitioning a $G'_{\mathcal{M}}$ into clusters involves building the cluster graph $G'^{c}_{\mathcal{M}} = \langle V, A \cup C \rangle$, where the number of clustering arcs in C is in $\mathcal{O}(|V|^2)$, as well as a finding the SCCs in $G'^{c}_{\mathcal{M}}$, which can be done in $\mathcal{O}(|V| + |A \cup C|) = \mathcal{O}(|V|^2 + |A|)$ time.

Resolving disjunctive edges, however, requires looping through disjunctive edges. Under the assumption that we never resolve any edge, every edge $\langle x, x' \rangle$ is visited once, and we search for a path from nodes in x to nodes in x' and paths in the opposite direction. Using breadth-first search or depth first search, this can be done in linear time for a given edge in a given disjunctive

Algorithm 3 Improved Algorithm for checking orientability.

1: **function** CHECK($G_{\mathcal{M}} = \langle V, A, X, E \rangle$)
2: Compute $G^c_{\mathcal{M}}$ as the cluster graph of $G_{\mathcal{M}}$.
3: Compute a topologically ordered list $P = \langle S_1, \ldots, S_n \rangle$ of SCCs of $G^c_{\mathcal{M}}$ using Tarjan's SCCs algorithm.
4: Let R be an empty list.
5: $\mathit{success} \leftarrow$ TRUE
6: **for** $i \leftarrow 1, n$ **do**
7: $\langle s', G_{\mathcal{M}}[S_i], R' \rangle$ CHECKPART($G_{\mathcal{M}}[S_i]$)
8: $\mathit{success} \leftarrow \mathit{success} \wedge s'$
9: Add $\langle s', G_{\mathcal{M}}[S_i], R' \rangle$ to the end of R
10: **end for**
11: **return** $\langle \mathit{success}, G_{\mathcal{M}}, R \rangle$
12: **end function**

13: **function** CHECKPART($G_{\mathcal{M}} = \langle V, A, X, E \rangle$)
14: RESOLVEEDGES($G_{\mathcal{M}}$)
15: **if** $G_{\mathcal{M}}$ has a cycle **then**
16: **return** $\langle$FALSE$, G_{\mathcal{M}}, \varnothing \rangle$
17: **else if** E is empty **then**
18: **return** $\langle$TRUE$, G_{\mathcal{M}}, \varnothing \rangle$
19: **else**
20: Choose a random disjunctive edge $\langle x_1, x_2 \rangle \in E$
21: **return** TRYBOTHDIRECTIONS($G_{\mathcal{M}}$, x_1, x_2)
22: **end if**
23: **end function**

24: **function** TRYBOTHDIRECTIONS($G_{\mathcal{M}} = \langle V, A, X, E \rangle$, x_1, x_2)
25: $\langle \mathit{success}, G', R' \rangle \leftarrow$ TRYDIRECTION($G_{\mathcal{M}}$, x_1, x_2)
26: **if** *success* **then**
27: **return** $\langle \mathit{success}, G', R' \rangle$
28: **end if**
29: $\langle \mathit{success}, G', R' \rangle \leftarrow$ TRYDIRECTION($G_{\mathcal{M}}$, x_2, x_1)
30: **if** *success* **then**
31: **return** $\langle \mathit{success}, G', R' \rangle$
32: **end if**
33: **return** $\langle$FALSE$, G_{\mathcal{M}}, \varnothing \rangle$
34: **end function**

35: **function** TRYDIRECTION($G_{\mathcal{M}} = \langle V, A, X, E \rangle$, x_s, x_t)
36: Copy $G_{\mathcal{M}}$ into $G'_{\mathcal{M}}$
37: ORIENT($G'_{\mathcal{M}}$, x_1, x_2)
38: **return** CHECK($G'_{\mathcal{M}}$)
39: **end function**

TU graph. Doing this for every disjunctive edge results in a complexity of $\mathcal{O}(|E| \cdot (|V|^2 + |A|))$ for resolving edges, which yields $\mathcal{O}(2^{|E|} \cdot |E| \cdot (|V|^2 + |A|))$ for the whole algorithm.

Note that this complexity is higher than the one of Algorithm 1. Yet, it is not clear whether there exists any disjunctive TU graph that results in the above scenario, namely where we are never able to apply partitioning or resolving edges, which results in considering all orientations of the original disjunctive TU graph. In particular, one should investigate whether the problem of checking consistency of a process model is NP-hard to understand whether a better algorithm can likely be found. Also, even if the scenario described above exists, it is questionable whether it will not occur in practice. In contrast, we expect that Algorithm 3 is well-suited for process models of real projects. In the following section, we present experimental results that support our expectations.

5.4.7 Experiments

To get an idea how the proposed algorithm performs in practice, we ran some experiments on the implementation. Specifically, we tested the algorithm on the hotel process model given in Fig. 4.1, as well as on three more process models which are slight variations of the hotel example. Thus, we consider four different process models:

consistent: This is the original hotel process model, which is consistent.

cycle: A variation of the hotel process model whose disjunctive TU graph is has cycles. In particular, the scope of the alternate precedence from Scaffolding Installation to Concrete Pouring is changed from *sector*, *level* to *sector*, which prevents Concrete Pouring from ever starting in the underground level. The diagram is shown in Fig. 5.14.

deadlock: A variation of the hotel process model that also inconsistent, but whose disjunctive TU graph is not cyclic. Specifically, the pattern shown in Fig. 5.11 is added to the hotel model and connected to Construction Site Preparation by a basic precedence at global scope. The diagram is shown in Fig. 5.15.

complex: A variation of the hotel process model that is also contains deadlocks, but whose disjunctive TU graph has many more nodes than the other models. Specifically, it has the same tasks as the *deadlock* model, but the tasks $T1, T2, T3, T4$ each contain all CUs of the interior phase instead of only two. The diagram is shown in Fig. 5.16.

To have some reference for the experiments, we measured the time needed by the NuSMV (Cimatti et al., 2002) model checker to check a process model encoded as a NuSMV module. For this purpose, we used the corresponding export functionality implemented in the application. The exported NuSMV module consists of two parts: *i)* a transition system, and *ii)* an LTL specification.

The transition system (TS) is a finite set of states over variables and a transition relation between states. In our case, it represents all possible well-defined executions on the set of events of the given process model. An LTL specification is used to encode the negation of all constraints in the model. NuSMV checks whether the LTL specification is valid in the given TS. If that is the case, the LTL specification is true for all possible path in the TS, which means that all possible well-defined executions violate some constraint in the model. Otherwise, NuSMV reports a counterexample that violates the LTL specification and corresponds to a well-defined execution that conforms to the model. The model is inconsistent iff the LTL specification is valid in the TS of all well-defined executions. Hence, we can use LTL model checking to determine whether the given process model is consistent.

All experiments are executed on a desktop PC with eight cores Intel i7-4770 of 3.40 GHz. For LTL model checking, we used the NuSMV model checker with Bounded Model Checking (BMC) (Biere et al., 2003). In BMC, one searches for a counterexample of the given specification in sequences of states $s_0, \dots, s_k$ whose length is bounded by an integer k. If no counterexample is found, k is increased until either a counterexample is found, k becomes too large, or it reaches a known upper bound.

For checking consistency of a process model $\mathcal{M}$ using BMC, we can define such an upper bound in terms of the number of nodes in the corresponding disjunctive TU graph $G_{\mathcal{M}} = \langle V, A, X, E \rangle$. Recall that every consistent model has a sequential well-defined execution that conforms to it.

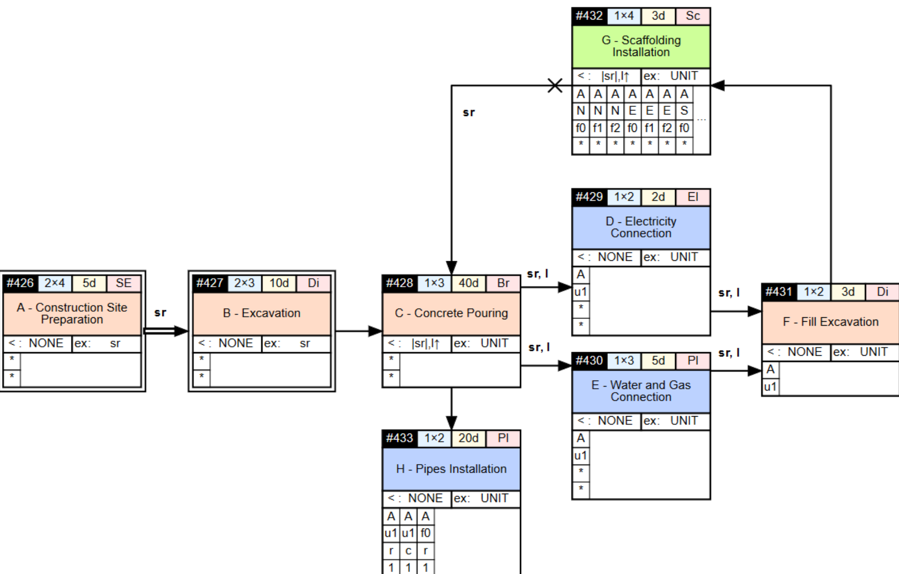


Figure 5.14: Variation of the hotel process model where the scope of the alternate precedence from Scaffolding Installation to Concrete Pouring is changed from *sector*, *level* to *level*, which leads to a cycle in the corresponding disjunctive TU graph.

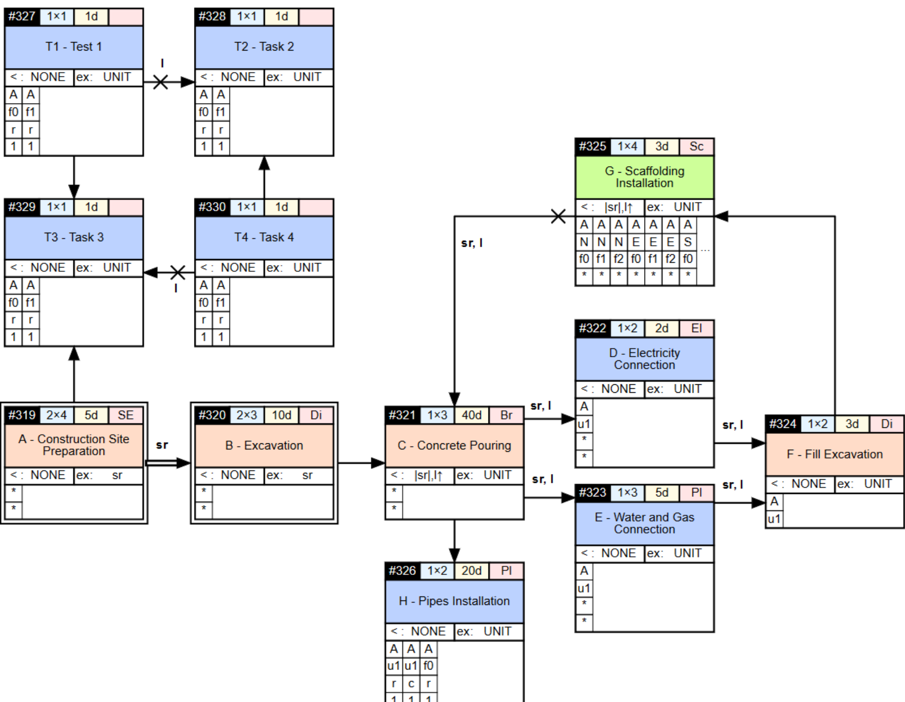


Figure 5.15: Variation of the hotel process model where the pattern shown in Fig. 5.11 is added to produce deadlocks.

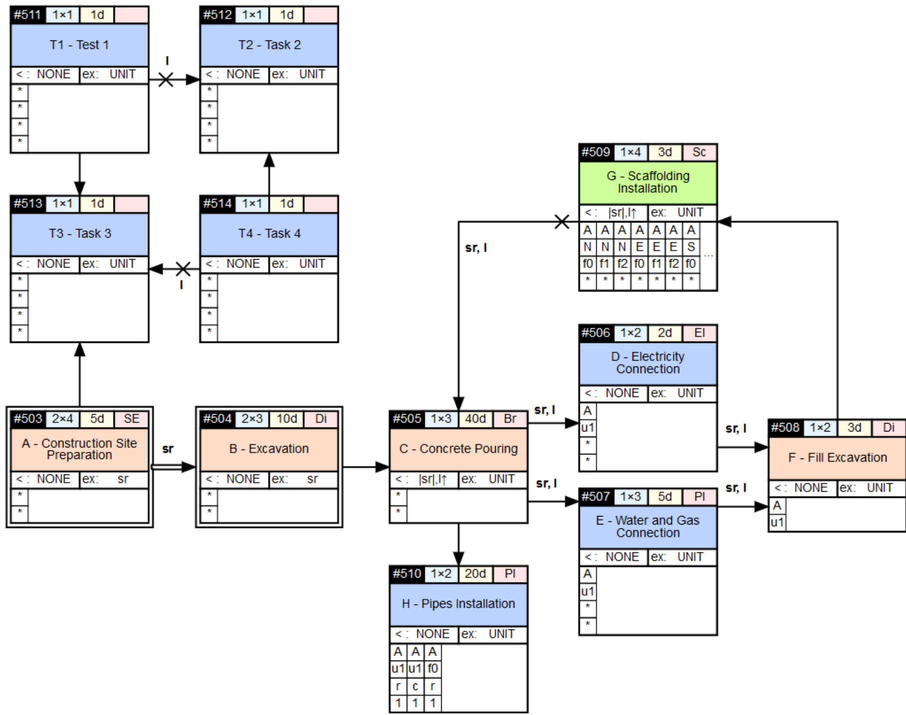


Figure 5.16: Variation of the hotel process model where a pattern similar to the one shown in Fig. 5.11, but where the tasks contain all CUs, is added to produce deadlocks similar to the process model shown in Fig. 5.15, but also producing many more nodes in the corresponding disjunctive TU graph.

Model	Tasks	Dep.	Nodes	Arcs	Edges (excl. simple)	Edges (all)	NuSMV
consistent	8	9	236	9415	231	524	2 min 35 s
cycle	8	9	236	10003	228	521	> 1 h
deadlock	12	14	244	9435	241	574	> 1 h
complex	12	14	424	15131	431	1740	> 1 h

Table 5.1: Quantities of the four process models considered in experiments.

Every sequential execution $v_1, \ldots, v_n$ can be shortened without violating any constraints in $\mathcal{M}$ by collecting the end event of v_i and the start event of the following node v_{i+1} into the same state, resulting in a sequence $s_0, \ldots, s_n$ where

$$\begin{aligned} s_0 &= \{\mathit{start}(v_1)\} \\ s_i &= \{\mathit{end}(v_i), \mathit{start}(v_{i+1})\} \quad \forall 0 < i < n \\ s_n &= \{\mathit{end}(n)\} \end{aligned}$$

Therefore, and because counterexamples in LTL model checking correspond to well-defined executions conforming to $\mathcal{M}$, we can use BMC with $k = |V|$ to check whether $\mathcal{M}$ is consistent.

Table 5.1 reports relevant quantities of the four process models, including the time needed for LTL model checking using NuSMV. The consistent model could be checked in about 150 seconds, whereas the other models took much more time, and we aborted the tests after one hour. It is not surprising that bounded model checking completes faster for consistent models than for inconsistent ones, considering that it actually tries to find an execution that conforms to the given model.

The experiments using the proposed algorithm were conducted on the same machine as the ones with NuSMV. Each of the four process models was tested and the times needed for building the disjunctive TU graph and for checking orientability of the resulting graph were measured separately. Moreover, to gain more insight on the benefit of the proposed strategies of ignoring simple edges, resolving edges, and partitioning the disjunctive TU graph, we tested each model with all combinations of these three strategies. Note that in the implementation, if simple disjunctive egdes are not ignored, the disjunctive TU graph also contains disjunctive edges corresponding to

Model	ISE	R	P	Translation	Check	Total
consistent				2 ms	27,981 ms	27,984 ms
consistent	✓			2 ms	6,546 ms	6,548 ms
consistent		✓		5 ms	181 ms	187 ms
consistent	✓	✓		2 ms	15 ms	17 ms
consistent			✓	4 ms	16 ms	21 ms
consistent	✓		✓	2 ms	8 ms	11 ms
consistent		✓	✓	5 ms	16 ms	21 ms
consistent	✓	✓	✓	7 ms	19 ms	27 ms
cycle				2 ms	< 1 ms	3 ms
cycle	✓			2 ms	1 ms	3 ms
cycle		✓		3 ms	1 ms	4 ms
cycle	✓	✓		1 ms	< 1 ms	2 ms
cycle			✓	2 ms	4 ms	6 ms
cycle	✓		✓	1 ms	3 ms	5 ms
cycle		✓	✓	2 ms	3 ms	6 ms
cycle	✓	✓	✓	1 ms	4 ms	5 ms
deadlock				5 ms	> 1 h	> 1 h
deadlock	✓			1 ms	22,751 ms	22,753 ms
deadlock		✓		3 ms	632 ms	636 ms
deadlock	✓	✓		1 ms	46 ms	48 ms
deadlock			✓	3 ms	29 ms	33 ms
deadlock	✓		✓	2 ms	18 ms	20 ms
deadlock		✓	✓	3 ms	8 ms	12 ms
deadlock	✓	✓	✓	2 ms	8 ms	10 ms
complex				5 ms	> 1 h	> 1 h
complex	✓			2 ms	> 1 h	> 1 h
complex		✓		5 ms	15,844 ms	15,849 ms
complex	✓	✓		3 ms	575 ms	579 ms
complex			✓	5 ms	> 1 h	> 1 h
complex	✓		✓	2 ms	33 ms	36 ms
complex		✓	✓	5 ms	59 ms	65 ms
complex	✓	✓	✓	2 ms	21 ms	23 ms

Table 5.2: Experimental results on four variations of the hotel process model with different combinations of using or not using the strategies of ignoring simple edges (ISE), resolving (R), and partitioning (P). Durations are rounded down to whole milliseconds.

implicit exclusiveness constraints. The results are presented in Table 5.2. Specifically, the reported durations represent the average time needed for three repetitions of the experiment, or indicate that a single execution of the experiment takes more than one hour.

We can summarize that, on average, the combination of all strategies results in the best performance. Only in the *cycle* variant, it is always a bit faster not to use partitioning, because partitioning is applied before checking for cycles. This suggests that we should perform a single additional check for cycles before partitioning to better handle these cases. On the other hand, that would in turn require a bit more time for checking disjunctive TU graphs that do not have cycles. Both partitioning and checking for cycles can be done efficiently, it never makes a big difference which of the two is done before the other. We decided to do partitioning before resolving to decrease the number of edges before trying to resolve them, and to only check for cycles while resolving.

The proposed strategies bring the most speedup in the presence of deadlocks, especially when the disjunctive TU graph becomes large. The *complex* model, in particular, can be checked in fractions of seconds as long as we use at least two out of the three strategies. Interestingly, on the disjunctive TU graph without simple edges, it is faster to use only partitioning rather than only resolving, whereas on the whole disjunctive TU graph, the converse is true.

Tasks	Dep.	Nodes	Arcs	Edges	Trans.	Check	Total
60	75	2,120	76,103	10,635	71 ms	526 ms	598 ms
120	173	4,240	168,681	42,470	106 ms	1,082 ms	1,189 ms
180	296	6,360	361,969	95,505	252 ms	3,429 ms	3,682 ms
240	452	8,480	478,701	169,740	425 ms	7,088 ms	7,513 ms
300	623	10,600	674,584	265,175	743 ms	13,455 ms	14,199 ms
360	822	12,720	948,099	381,810	2,239 ms	21,983 ms	24,223 ms
420	1,044	14,840	1,309,129	519,645	2,960 ms	37,398 ms	40,359 ms
480	1,291	16,960	1,436,759	678,680	3,846 ms	52,020 ms	55,866 ms
720	2,562	25,440	3,082,925	1,526,820	7,964 ms	371,445 ms	379,409 ms
960	4,187	33,920	5,217,426	2,714,160	27,509 ms	OOM	OOM

Table 5.3: Experimental results for process models with an increasing number of tasks. The number of tasks is increased by creating several copies of the complex variant of the hotel example. Tasks of different copies are connected by randomly generated basic precedences at global scope. With 80 copies of the original process model, the implementation runs out of memory (OOM).

We showed that the implementation of the proposed algorithm outperforms the NuSMV model checker, and that all of the proposed strategies indeed improve the algorithm's performance. Now we want to understand the limits of the proposed algorithm. One way to do this is to increase the number of tasks. For this purpose, we generated larger models by taking the *complex* variant of the hotel example shown in Fig. 5.16, copied it several times, and added basic precedences at global scope between tasks of one copy to tasks of another. To not introduce cycles, because they can be checked easily, we defined an order on the copies of the model and only added precedences following that order. Moreover, to get a model of a realistic shape, we defined a ratio of 1.1 of number of dependencies between models and the number of models. Also, to effectively increase the size of the corresponding disjunctive TU graph, we must give fresh names to the activities of the copied process. Otherwise, all copies of a task would yield the same nodes.

We started by replicating the complex hotel process model five times and kept adding more copies of it until we reached a total of 80 copies of the original process model. Table 5.3 summarizes the results of experiments on the implementation using all the proposed strategies. In particular, we now always ignore simple edges, so they are also excluded in the reported numbers of disjunctive edges in the table. With 80 copies of the original model, i.e. 960 tasks, the implementation runs out of memory. Indeed, with over five million arcs and two million disjunctive edges, the corresponding disjunctive TU graph becomes very large. Still, it is not clear whether the cause of running out of memory is in the algorithm or only in its implementation. Specifically, the implementation uses auxiliary data structures which are used in the implementation to make the algorithm faster. For example, the disjunctive TU graph implementation maintains a map of nodes to the disjunctive edges in which they are involved. The main advantage of this is that while one disjunctive edge $\langle x, x' \rangle$ is tried to be resolved, for instance from x to x', we can efficiently resolve other disjunctive edges between x and the nodes that we find on the way. However, if many nodes are involved in many disjunctive edges, this requires a lot of memory.

On the other hand, one may wonder whether a process model of 960 tasks is realistic. Already for 720 tasks, the algorithm completes in approximately six and a half minutes, which is still an acceptable amount of time to wait with respect to the large size of the process model. Probably, 300 is a more realistic but still high number of tasks, which the implementation is able to handle within 15 seconds. In all process models, there are in total 319 CUs for all phases, where 268 belong to the exterior phase, 47 belong to the interior phase, and 4 belong to the skeleton phase. This suggests that we can handle about 700 tasks and 300 CUs in approximately six minutes, and 300 tasks in 15 seconds. Yet, most of the tasks only contain some of the CUs, so the resulting number of nodes in the disjunctive TU graph is a better indication of the size of the model. Therefore, assuming that the shape of the model is realistic, we can say that the implementation is able to handle about 30,000 nodes within seven minutes, and 10,000 nodes within 15 seconds.

By increasing the number of tasks, we pushed the algorithm to the limit in terms of required memory, while the performance was still acceptable given the large diagrams. Thus, to understand the limits of the algorithm in terms of execution time, we need to construct the process model more carefully to make it hard to check. In the examples with the replicated models, we can always apply

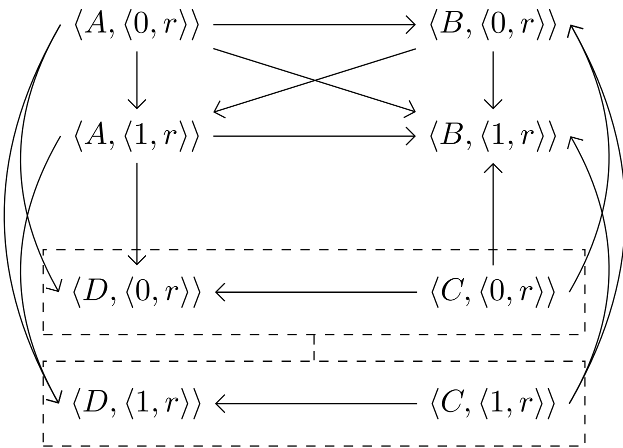


Figure 5.17: The disjunctive TU graph that results from choosing one direction of a disjunctive edge in Fig. 5.11(b).

partitioning to split the whole disjunctive TU graph into several disjunctive TU graphs per model. The subgraphs are the same for all models. After the initial partitioning operation, the algorithm performs the same amount of work for each copy of the original model, and then combine the results. Furthermore, the implementation checks subgraphs in parallel. Depending on the number of cores available in the executing machine, this can give some significant performance boost.

Thus, if increasing the number of tasks only increases the number of subgraphs, the required time does not grow very fast. Instead, for making the time explode, we must come up with a process model that results in a single subgraph and then increase the complexity of that subgraph. Moreover, we should also think about the structure of subgraph such that it becomes hard to check. For this purpose, recall that in Section 5.4.6, we considered a hypothetical worst-case scenario of an inconsistent process model where the algorithm can neither apply partitioning nor resolving, and only finds cycles after guessing a direction for almost all disjunctive edges. In such a case, the algorithm would consider almost all directions for all disjunctive edges. This scenario, however, is only hypothetical; we do not know whether any process model exists resulting in the described behavior. Therefore, we use a process model that comes as close as possible, i.e. where partitioning and resolving can be applied only rarely.

Table 5.2 suggests that process models containing deadlocks are the hardest to check. Indeed, deadlocks can only be detected by trying out both possible directions for at least one disjunctive edge. As an example, consider Fig. 5.11(b), which shows a disjunctive TU graph $G_{\mathcal{M}}$ with deadlocks. In particular, it shows the pattern that is used to produce deadlocks also in the *deadlock* and *complex* process models shown in Fig. 5.15 and Fig. 5.15, respectively. Note that $G_{\mathcal{M}}$ only has a one single cluster, thus it cannot be partitioned into smaller subgraphs. After partitioning, the algorithm tries to resolve edges, but does not succeed, because for each disjunctive edge $\langle x, x' \rangle$, there is no path between a node in x and a node in x'. Therefore, one disjunctive edge is chosen and oriented in some direction. Suppose we choose the edge e_1 between the nodes of activities A and B and orient it such that the two activities must finish in level 0 before they can start in level 1, which results in the disjunctive TU graph shown in Fig. 5.17.

Still, the resulting graph cannot be partitioned, because both nodes of activity C can reach both nodes of activity D. For the very same reason, however, it is possible to resolve the remaining disjunctive edge e_2 in both directions. This means that a cycle is introduced, which is then detected. Then, the other direction of e_1 is tried out, and again e_2 is resolved and a cycle is detected, so the algorithm returns that $G_{\mathcal{M}}$ is not orientable.

In the described example, there are only two disjunctive edges. To make it harder to check, we can increase the number of disjunctive edges. In the *complex* process model shown in Fig. 5.16, we do this by requiring the four tasks to be executed in all CUs instead of only two. Because the alternate precedence is defined at the scope *level*, and there are four different levels in the process model, each of the two alternate precedences produces four exclusive groups which are pairwise

CUs	Nodes	Arcs	Edges	Translation	Check	Total
50	200	5,100	2,450	11 ms	954 ms	966 ms
100	400	20,200	9,900	13 ms	10,926 ms	10,940 ms
150	600	45,300	22,350	20 ms	55,509 ms	55,530 ms
200	800	80,400	39,800	67 ms	213,980 ms	214,048 ms
300	1200	180,600	89,700	105 ms	1,072,927 ms	1,073,032 ms
400	1600	320,800	159,600	201 ms	> 1 h	> 1 h

Table 5.4: Experimental results for an increasingly complex process model with deadlocks. The process model is similar to Fig. 5.11, but all tasks are executed in all CUs, and the scope of the alternate precedence is at the granularity of CUs. The complexity of the model is increased by increasing the number of CUs.

connected by disjunctive edges. In this case, this results in a total of twelve disjunctive edges in the deadlock pattern. To further increase the number of disjunctive edges, we can refine the scope to the granularity of a single CU, i.e. to the unit scope. For n CUs, we get $n(n-1)$ disjunctive edges.

We ran experiments starting with $n = 50$ and increased n up to 600. Table 5.4 shows the results of these experiments. Indeed, the execution time grows very fast with the number of CUs. To check consistency of the given pattern, the implementation requires several seconds for 100 CUs, several minutes for between 200 and 300 CUs, and over one hour for 400 CUs. We consider these results acceptable, given that we specifically designed the process model to be hard to check. Also, one may again ask whether the tested process models are realistic. Specifically, it is questionable whether the used pattern will occur in practice with alternate precedences at such a fine scope and hundreds of CUs. Assuming that this is not the case, we conclude that the algorithm performs sufficiently well in practice.

Chapter 6

On Scheduling

The main goal of the theoretic part of this thesis is to develop algorithms for checking consistency of process models. For that purpose, in Chapter 5, we introduced the disjunctive TU graph, a graph representation that is able to capture all constraints of a given process model. In this section, we show that this representation of a model is useful not only for checking consistency, but also for solving other problems more related to scheduling. Specifically, we investigate checking conformance of an existing schedule to a process model, generating a schedule conforming to a given model, and computing the critical path of a model.

6.1 Checking Conformance

We can use the disjunctive TU graph representation $G_{\mathcal{M}}$ of a process model $\mathcal{M}$ to check conformance of an execution to $\mathcal{M}$. In contrast to checking consistency, however, it is not sufficient to check whether all constraints *can* be satisfied. Instead, we must check whether the given execution actually *does* satisfy all constraints. Therefore, for checking conformance, we must capture *all* constraints in a model, including simple disjunctive edges, and in particular implicit exclusiveness constraints.

To capture implicit exclusiveness constraints, for all tasks $\mathtt{t}$ that have no explicit exclusiveness constraint, we assume an exclusiveness constraint at a scope $\mathtt{s}$ that contains all attributes in the phase of the given task and add the corresponding exclusive groups and disjunctive edges. Specifically, this results in an exclusive group $\{v\}$ for each node $v \in V(\mathtt{t})$, and a disjunctive edge $\langle\{v\}, \{v'\}\rangle$ from these exclusive groups to each other node v' that has the same values for all attributes in $\mathtt{s}$, i.e. where $\Pi_{\mathtt{s}}(v') = v$.

Once we have a disjunctive graph representation $G_{\mathcal{M}}$ that captures all constraints in a model $\mathcal{M}$, we can check conformance of an execution to $\mathcal{M}$ by traversing in parallel the states of the execution and the nodes of $\mathcal{G}_{\mathcal{M}}$ and check if all the events of a state are allowed to occur based on the previous events. In particular, we label the nodes of $G_{\mathcal{M}}$ in four possible ways:

blocked The node has not started yet and is not allowed to start.

ready The node has not started yet but is allowed to start.

started The node has started but is not finished yet.

ended The node finished its execution.

At the beginning, nodes that do not have any predecessor node are *ready*, while all other nodes are *blocked*. A *ready* node becomes a *started* node when the corresponding start event occurs in the schedule. Similarly, a *started* node becomes an *ended* node when the corresponding end event occurs. A *blocked* node becomes *ready* when all its predecessor nodes have *ended*.

Furthermore, we also need to consider disjunctive edges. Take a disjunctive edge $e = \langle x, x' \rangle$. When a node $v \in x$ is started in a state before starting any node in x', the semantics of disjunctive edges require that all nodes in x must end before all nodes in x' start. Therefore, when encountering v, we can already resolve e in the direction $x \to x'$. Because this introduces new arcs to the nodes of x', all nodes in x' become *blocked*, even if they were *ready* before. Accordingly, if a node $v' \in x'$

is encountered before any node of x, e is oriented in the direction $x' \rightarrow x$, and all nodes in x become blocked.

Finally, we need to pay attention to the details of which start and end events are allowed to occur in a single state. First, for a given node v, the end event *end*(v) must occur strictly after the start event *start*(v). In particular, it is not possible that both occur in the same state. Second, for a precedence $v \rightarrow v'$, denoted by an arc in the disjunctive TU graph, *start*(v') must not occur before *end*(v), but both may occur in the same state. We can capture these two constraints by considering all end events in a state before all start events. When the state contains both a start and an end event of the same node, the end event is registered before the start event, which correctly yields a problem. Similarly, if there is an arc $v \rightarrow v'$, and both *end*(v) and *start*(v') occur in the same state, we first consider the end event of v and thus correctly conclude that the precedence from v to v' is satisfied.

Algorithm 4 shows the detailed procedure for checking conformance that results from all these considerations. For simplicity, we use $pre(v)$ to denote the predecessors and $suc(v)$ to denote the successors of a node v, where for a disjunctive TU graph $G_{\mathcal{M}} = \langle V, A, X, E\rangle$:

$$pre(v) := \{v' \in V \mid \langle v', v\rangle \in A\}$$
$$suc(v) := \{v' \in V \mid \langle v, v'\rangle \in A\}$$

Algorithm 4 Conformance of a schedule to a disjunctive TU graph.

```
 1: function CONFORMS(G_M = ⟨V, A, X, E⟩, s = s_1, ..., s_n)
 2:     V_ready ← {v ∈ V | pre(v) = ∅}
 3:     V_started ← ∅
 4:     V_ended ← ∅
 5:     for i ← 1, n do

 6:         for all end(v) ∈ s_i do                    ▷ Consider end events first
 7:             if v ∉ V_started then
 8:                 return false
 9:             end if
10:             V_started ← V_started \ {v}
11:             V_ended ← V_ended ∪ {v}
12:             for all v' ∈ suc(v) do
13:                 A ← A \ {v → v'}
14:                 if pre(v') = ∅ then
15:                     V_ready ← V_ready ∪ {v'}
16:                 end if
17:             end for
18:         end for

19:         for all start(v) ∈ s_i do                  ▷ Consider start events last
20:             if v ∉ V_ready then
21:                 return false
22:             end if
23:             for all e = ⟨x, x'⟩ ∈ E such that v ∈ x do
24:                 G_M ← ORIENT(G_M, x, x')
25:                 V_ready ← V_ready \ x'              ▷ Block all target nodes
26:             end for
27:         end for

28:     end for

29:     return V_ended = V
30: end function
```

6.2 Generating Schedules and CPM

Algorithm 5

```
 1: function findAcyclicOrientation(G_M = ⟨V, A, X, E⟩)
 2:     ⟨success, G_M, R⟩ ← check(G_M)
 3:     if ¬success then
 4:         return "Failure"
 5:     end if
 6:     L = ⟨G_1, ..., G_n⟩ ← flattenResult(⟨success, G_M, R⟩)
 7:     for i ← 1, n do
 8:         ⟨V_i, A_i, E_i⟩ ← G_i
 9:         for all v ∈ V_i do
10:             leaf(v) ← i
11:         end for
12:         Add all A_i to A
13:     end for
14:     for all ⟨x, x'⟩ ∈ E do
15:         Choose random v ∈ x and random v' ∈ x'
16:         if leaf(v) < leaf(v') then
17:             orient(G_M, x, x')
18:         else if leaf(v') < leaf(v) then
19:             orient(G_M, x', x)
20:         end if
21:     end for
22:     return G_M
23: end function

24: function flattenResult(r = ⟨success, G_M, R⟩)
25:     Let L be an empty list
26:     if R is empty then
27:         Add G_M to L
28:     else
29:         for all r' ∈ R do
30:             Let L' ← flattenResult(r')
31:             Append L' to L
32:         end for
33:     end if
34:     return L
35: end function
```

In Section 5.3, we presented Algorithm 3, which checks consistency of a model by checking whether the corresponding disjunctive TU graph has an acyclic orientation. If it turns out to be consistent, it returns a tree of subgraphs that allows to efficiently construct an actual acyclic orientation. We can use this orientation to produce a schedule that conforms to the model.

For example, topologically ordering the nodes in the acyclic orientation results in a linear execution that conforms to the model. Yet, an execution obtained in this way might be far from optimal with respect to total duration, workflow continuity, resource usage, and costs.

One idea for optimization is to integrate principles and algorithms from the Location-Based Management System (LBMS) (Kenley and Seppänen, 2006). In the LBMS, the process is modeled using *layered CPM logic*, which allows to express precedences, lead times and lag times among tasks and locations at various levels of granularity in the location Breakdown Structure (LBS) (Kenley and Seppänen, 2006, chap. 5, pg. 133-144). Evaluating the logic yields a Critical Path Method (CPM) network. In addition, it allows to indicate for each task whether it is more important to execute it continuously or as soon as possible. To compute a schedule that satisfies the layered CPM logic, the LBMS defines an extended CPM algorithm that operates on the resulting network and

$$\begin{array}{ccc} \langle \mathsf{t}, \mathsf{u}_1 \rangle & \text{--} & \langle \mathsf{t}', \mathsf{u}_1 \rangle \\ \downarrow & & \downarrow \\ \langle \mathsf{t}, \mathsf{u}_2 \rangle & \text{--} & \langle \mathsf{t}', \mathsf{u}_2 \rangle \end{array}$$

(a) A disjunctive TU graph of a simple process model.

$$\begin{array}{ccc} \langle \mathsf{t}, \mathsf{u}_1 \rangle & \rightarrow & \langle \mathsf{t}', \mathsf{u}_1 \rangle \\ \downarrow & & \downarrow \\ \langle \mathsf{t}, \mathsf{u}_2 \rangle & \leftarrow & \langle \mathsf{t}', \mathsf{u}_2 \rangle \end{array}$$

(b) An orientation of the disjunctive TU graph that requires to execute all nodes sequentially.

$$\begin{array}{ccc} \langle \mathsf{t}, \mathsf{u}_1 \rangle & \rightarrow & \langle \mathsf{t}', \mathsf{u}_1 \rangle \\ \downarrow & & \downarrow \\ \langle \mathsf{t}, \mathsf{u}_2 \rangle & \rightarrow & \langle \mathsf{t}', \mathsf{u}_2 \rangle \end{array}$$

(c) An orientation of the disjunctive TU graph that permits parallelizing $\langle \mathsf{t}, \mathsf{u}_2 \rangle$ and $\langle \mathsf{t}', \mathsf{u}_1 \rangle$.

Figure 6.1: A disjunctive TU graph with two possible orientations that allow different degrees of parallelism.

takes into account whether tasks should be executed continuously or not (Kenley and Seppänen, 2006, chap. 5, pg. 147-156).

The generated CPM networks in LBMS are directed graphs with some extra information such as expected durations and workflow continuity constraints. Given an acyclic orientation of the disjunctive TU graph of a model, we can get the duration of the execution of a node from the model. Indeed, a process model defines the pitch for each task, that is the number of units that can be performed on a working day. From this information, we can derive the duration for a TU-node. The result is a directed graph where arcs correspond to precedences and nodes are labeled with durations, i.e. a graph corresponding to an activity network as expected by CPM. To use the extended CPM algorithm of the LBMS, we can add workflow continuity constraints to define whether a task must be executed continuously, or whether it should start in a location as soon as possible. As a result, we get a schedule that conforms to our model and which is optimized with respect to workflow continuity. Moreover, the algorithm computes the location-based equivalent to free float and total float in the resulting network. Free float is the amount of time an activity can be delayed without delaying the next activity, whereas total float is the amount of time an activity can be delayed without delaying the whole project. These two measures provide further insights useful for project control.

If multiple acyclic orientation of the disjunctive TU graph exists, the quality of the resulting schedule is largely dependent on the particular orientation that is chosen. As an example, consider Fig. 6.1. It shows a disjunctive TU graph in Fig. 6.1(a) corresponding to a process model that contains two tasks t, t' which both have two CUs $\mathsf{u}_1, \mathsf{u}_2$, and an ordering constraint that requires u_1 to be executed before u_2 for each task. None of the two tasks has an explicit exclusiveness constraint, so only the implicit exclusiveness constraints at the unit scope apply, which we also consider because we need to build an actual schedule. These implicit exclusiveness constraints result in two disjunctive edges between nodes of the same CU. Hence, there are four possible orientations of the disjunctive TU graph, all of which are acyclic. Figure 6.1(b) shows one of these orientations G that requires to execute the nodes one after the other. Consequently, the sum of the durations of all the nodes represents a lower bound of the total duration of any schedule. More precisely, the duration of a schedule is equal to the sum of the durations of all nodes plus the sum of all delays between the execution of nodes. Beause we are only interested in schedules which are optimal with respect to time, we can assume there are no delays. That is, if $d(v)$ denotes the duration of a node and D_{seq} is the total duration of the resulting schedule, then we have

$$D_{seq} = d(\mathsf{t}, \mathsf{u}_1) + d(\mathsf{t}, \mathsf{u}_2) + d(\mathsf{t}', \mathsf{u}_1) + d(\mathsf{t}', \mathsf{u}_2)$$

Differently, the orientation G' shown in Fig. 6.1(c) allows to execute $\langle \mathsf{t}, \mathsf{u}_2 \rangle$ and $\langle \mathsf{t}', \mathsf{u}_1 \rangle$ in parallel. Therefore, if D_{par} is the total duration of a schedule derived from this orientation, we have

$$D_{par} = d(\mathsf{t}, \mathsf{u}_1) + \max(d(\mathsf{t}, \mathsf{u}_2), d(\mathsf{t}', \mathsf{u}_1)) + d(\mathsf{t}', \mathsf{u}_2)$$

If we further assume that the expected duration of each node is greater than zero, we get

$$D_{par} < D_{seq}$$

Hence, if one wants to generate a schedule with optimum total duration, it is not sufficient to just consider one arbitrary acyclic orientation.

Recall that our disjunctive TU graph representation is inspired from the disjunctive graph representation, which is used to represent instances of jobshop scheduling problems. There is a large literature on exact and approximate algorithms for solving the jobshop scheduling problem using disjunctive graph (Błażewicz et al., 1996; Brucker et al., 1994; Abdelmaguid, 2009; Van Laarhoven et al., 1992). We could take inspiration from these algorithms and adapt them to our graph representation. In particular, Van Laarhoven et al. (1992) show that the search space of optimal orientations for a jobshop scheduling problem can be limited to those that can be produced from another acyclic orientation of the corresponding disjunctive graph by reversing the direction of a disjunctive edge that forms a *critical path* in the previous orientation. A critical arc is an arc that is on a critical path, i.e. on a longest path in the graph. The reason is that for an orientation $\langle V, A \cup A' \rangle$ of a disjunctive graph $\langle V, A, E \rangle$, *i)* reversing a non-critical arc $\langle v, v' \rangle \in E, A'$ can only introduce a new longest path, but not reduce the current longest path, and *ii)* reversing a critical arc $v \to v'$ always leads to an acyclic orientation, because otherwise there would be path from v to v' that is longer than $v \to v'$ since A' and A are disjoint, so $v \to v'$ cannot be critical.

While *i)* also holds for the disjunctive TU graph representation of process models, *ii)* does not hold, because it is possible that two nodes are connected both by conjunctive arcs and disjunctive edges. Also, recall that disjunctive edges in our representation do not connect two single nodes but two sets of nodes, and an orientation into one direction of the edge corresponds to arcs of all nodes of one set to all nodes of the other set. When reversing such a disjunctive edge, we need to reverse all corresponding arcs at once. The overall argument that we can reduce the search space by only reversing critical arcs, however, is still valid. The difference is that in our approach, that search space also includes cyclic orientations.

6.3 Scheduling Complexity

In Section 6.2, we investigated how to generate a schedule and use location-based extensions of CPM algorithms from a process model based on an acyclic orientation of the corresponding disjunctive TU graph. In particular, the location-based CPM algorithm is able to optimize not only the total duration, but also workflow continuity.

In this section, we show that the problem of finding an optimal schedule with respect to the total execution time is NP-hard. Further, we show that the approach discussed in Section 6.2 is not able to find the optimal solution because it only investigates one arbitrary orientation.

We prove NP-completeness of the problem of finding an optimal schedule by reduction from the flowshop scheduling problem. The flowshop scheduling problem is a special class of the jobshop scheduling problem, and is known to be NP-complete (Garey et al., 1976). In a flowshop, there is a set of n jobs $\{J_1, \dots, J_n\}$ and a set of m machines $\{M_1, \dots, M_m\}$. Each job J_i consists of m operations $O_{1,1}, \dots, O_{1,m}$, one for each machine. Each operation $O_{i,j}$ can only be processed by the corresponding machine M_j, and each machine can only process one operation at a time. Moreover, for each job J_i, operation $O_{i,j}$ must be processed before processing operation $O_{i,j+1}$ for all $1 \leq j < m$. Finally, for each operation $O_{i,j}$, the duration $d_{i,j}$ is known. The problem is to find a schedule with optimal *makespan*, i.e. to find a starting time $s_{i,j}$ for each operation $O_{i,j}$ such that all the constraints above are satisfied and where the largest finish time

$$f_{i,j} := s_{i,j} + d_{i,j}$$

of all operations $O_{i,j}$, is minimal.

This is an optimization problem, so we need to rephrase it as a decision problem before we can provide a reduction. The decision problem version of the flowshop scheduling problem is as follows: Given a threshold T, a set of jobs, a set of machines, a set of operations, and an execution time for each job, is there a schedule such that the makespan does not exceed T?

Accordingly, we phrase the decision version of the process model scheduling problem as follows: Given a threshold T and a process model $\mathcal{M}$, is there a schedule that conforms to $\mathcal{M}$ where the

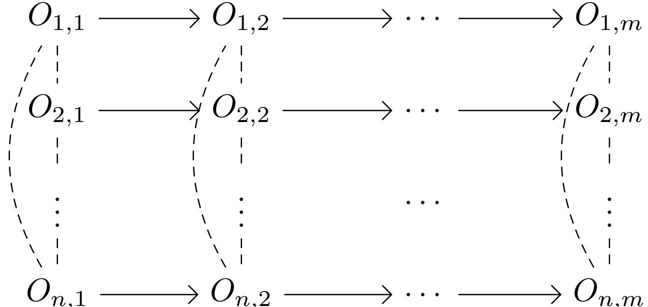


Figure 6.2: The disjunctive graph representation of a flowhop scheduling problem with n jobs and m machines.

expected total execution time for each task is equally distributed among all CUs such that the difference between the first start date and the last end date does not exceed T?

We now show NP-completeness by providing a reduction of the decision version of the flowshop scheduling problem to the decision version of the process model scheduling problem. Recall that disjunctive TU graphs in our approach are an extension of disjunctive graphs, which can be used to graphically represent jobshop scheduling problems, and therefore also flowshop scheduling problems. Figure 6.2 shows the disjunctive graph representation of a flowshop of n jobs and m machines. Each job corresponds to a horizontal chain of operations, where conjunctive arcs indicate the ordering on the operations. Operations that are to be executed by the same machine are connected by disjunctive edges. Hence, for each machine, the operations executed on that machine form a clique.

The idea of the reduction is that for any given flowshop scheduling problem, we can construct a process model such that the corresponding disjunctive TU graph is structurally identical to the disjunctive graph representation of the flowshop scheduling problem.

Theorem 7. *The process model scheduling problem is NP-complete.*

Proof. We reduce the problem of whether for a given flowshop with n jobs and m machines, there exists a schedule with a makespan that does not exceed a given threshold T, to the problem of determining whether a given process model has a schedule that takes at most T days. Let $\{J_1, \ldots, J_n\}$ be the set of jobs, $\{M_1, \ldots, M_m\}$ the set of machines, $\{O_{i,1}, \ldots, O_{i,m}\}$ the set of operations for each job J_i, and $d_{i,j}$ the execution time for each operation $O_{i,j}$, for all $1 \leq i \leq n$, $1 \leq j \leq m$. We will define a process model $\mathcal{M}$ such that for a given threshold T, the flowshop has a schedule whose execution time that does not exceed T iff the process model $\mathcal{M}$ has a conforming schedule whose execution time also does not exceed T. We do this by mapping operations to tasks machines to CUs, and by adding precedences between tasks corresponding to operations of the same job to capture the ordering constraints among them.

First, we define an attribute *machine* with a range that contains all the machines $\{M_1, \ldots, M_m\}$, and a phase whose CU relation is equal to *machine*, i.e. the relation consists of the single attribute *machine* and contains all of its values. Next, for each operation $O_{i,j}$, we define an activity $T_{i,j}$ of the phase *machine* and a task box $\mathtt{t}_{i,j}$ with a single CU M_j and an expected duration of $d_{i,j}$. The task boxes have no ordering constraints, and only implicit exclusiveness constraints at the unit scope. Finally, for all $1 \leq i \leq n$, $1 \leq j < m$, we add a (basic) precedence constraint at the global scope from $\mathtt{t}_{i,j}$ to $\mathtt{t}_{i+1,j}$.

Clearly, the dependencies capture the ordering on operations of the same job. Further, implicit exclusiveness constraints at the unit scope of all task boxes capture the requirement that no machine can process multiple operations at the same time. Thus, the flowshop and the process model $\mathcal{M}$ carry the same execution semantics. Because each task box has the same duration as the corresponding operation, we get that there is a flowshop schedule of makespan D iff there is a schedule where D is the difference between the first start date and the last end date. Hence, solving the problem for $\mathcal{M}$ also solves the problem for the flowshop. $\square$

Chapter 7

Application

In Chapters 4 and 5, we covered the theoretic part of this thesis by discussing the process modeling formalism and algorithms how to check whether a process model is consistent. In this chapter, we focus on the application that we developed to support the design of process models.

First, we give the requirements that we considered for the development of the application. Then, we present the architecture and the design we chose for meeting those requirements, as well as some implementation details such as chosen libraries and technologies. Finally, we report a case study that we conducted in a planning workshop of a real company where we tested the implemented application and thus the proposed extension of the PRECISE modeling language.

7.1 Requirements

The purpose of the applications is to support process modeling in PRECISE. This shall be achieved through two interfaces: a graphical and a textual one. The main one is the graphical user interface (GUI). It lets users interact with process models in a user-friendly way. Additionally, the system shall provide a textual interface in order to be usable and testable independently of the GUI. Finally, the GUI shall integrate the textual interface as an import and export feature.

7.1.1 Functional Requirements

Graphical User Interface The high-level use cases to be supported by the GUI are shown in Fig. 7.1. To manage a list of all the models stored in the system, the user must be able to *create* and *delete* models. For quickly creating a model that is similar to an existing one, it should also be possible to *duplicate* a model. Alternatively, the user can create a model by *importing* an existing model from the system or from a local file, using the textual representation of models. Accordingly, a textual representation of the process model can also be *exported* as a file that can be saved locally, which can be imported later. Moreover, the application shall also allow to produce an excel sheet with a list of tasks and their estimated costs in terms of man-hours.

The use cases *edit model* and *inspect model* concern the interaction with a single model and are specified in more detail in Fig. 7.2 and 7.3, respectively, and in the following paragraphs.

Edit Model Recall that a process model consists of a configuration part and a flow part. To support collaborative modeling, the flow part must be defined using interactive process diagrams represented in the graphical language defined in Chapter 4. This is the most crucial part of the system.

In the diagram, a user shall be able to specify all the information that belongs to the flow. This includes *adding* and *deleting* tasks and dependencies as well as *editing* their details.

The details of tasks are the task's activity, the pitch parameters, CUs, and the ordering and exclusiveness constraint. The pitch parameters are the estimated total duration of a task, the planned number of crews, the optimal crew size, the total quantity of work to be performed in the unit of measure specified by the activity of the task, and the estimated crew productivity, i.e. the estimated progress of each crew per day. Note that except for the crew size, these parameters depend on each other. Therefore, if only one of them is missing, the application shall calculate it

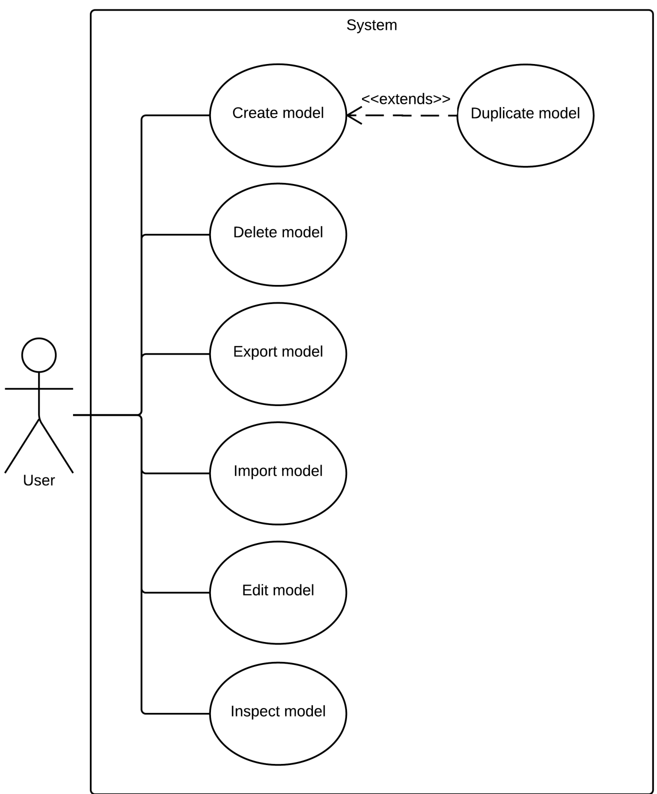


Figure 7.1: High-level use cases.

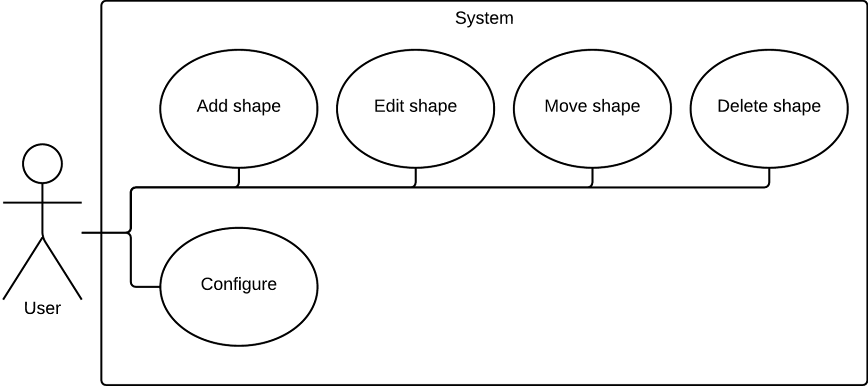


Figure 7.2: Use cases for editing models.

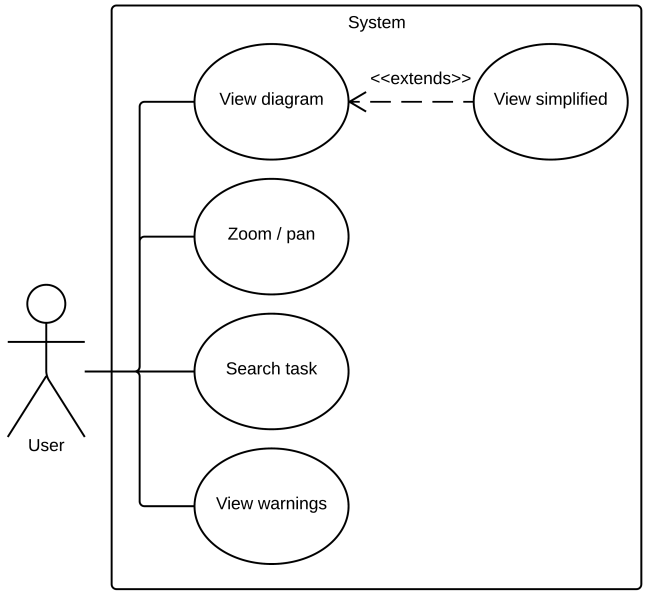


Figure 7.3: Use cases for inspecting models.

based on the others. Also, if all of them are specified, the application shall check whether they are consistent among each other. Finally, the application should also compute the resulting total man-hours required for performing the task.

The details of dependencies are the scope and the kind, i.e. whether it is a basic precedence, an alternate precedence, a chain precedence, or both an alternate and a chain precedence. When specifying pitch parameters, the system shall calculate missing parameters based on other parameters, if possible. Moreover, it must be possible to control the positions of tasks and the shape of dependencies.

The configuration part is typically known beforehand, so it can be defined before the meeting, which can be done outside the diagram. One exception to this are activities. It must be possible to define activities on the fly from the diagram, because this is often needed during workshops.

Inspect Models The goal of a process modeling workshop is to define a process model. Thus, it is crucial that the application support creating and editing process models. However, presenting the currently modeled process in an effective way is equally important. In Fig. 7.3, we define what effective presentation actually means by decomposing it into several use cases.

First, users need to be able to *view* a diagram of the modeled process. To allow collaborative modeling, this diagram should show all the information relevant for the execution, i.e. tasks with CUs and ordering and exclusiveness constraints, and dependencies with their kind and scope. On the other hand, the more information there is, the more cluttered the diagram becomes, and the harder it becomes to understand the big picture. Therefore, the system shall provide a *simplified* view where some details are hidden.

Similarly, when diagrams get big, it becomes difficult to find some particular tasks. Therefore, the system shall allow to highlight all the tasks that match some criteria provided by the user, e.g. the name of an activity, or the name of a craft.

Further, the number of tasks in a diagram can vary. Nonetheless, users want to see the whole diagram on one screen on the one hand, and read all the information of a task on the other hand. To support this, the system the user shall be able to control the viewport by zooming and moving the paper.

Finally, the system shall help the user to define models that are consistent and that do not lack any important information. If there are such problems, the system shall provide a descriptive error message and highlight the corresponding components in the diagram.

Textual representation To make the application usable and testable even without a GUI, it should provide an alternative interface that is based on a textual representation of process models. In particular, the system shall be able to produce a text file of an existing model as well as to parse a text file and import it as a new model.

For any errors encountered during parsing and interpreting the file, a descriptive error message should be returned. The format of the textual representation shall be ease to read by humans and easy to parse by machines.

7.1.2 Nonfunctional Requirements

Usability Considering that the diagram is defined collaboratively by many people, in a workshop of fixed duration, the interface of defining diagrams should be as simple as possible. To this aim, a graphical representation seems to be appropriate to support the discussion among the participants. The configuration part, however, is usually defined by a single person before the collaborative workshops starts. Moreover, as opposed to the diagram, the information to be inserted into the system is known in advance. For this purpose, a simpler interface that allows one to insert information quickly is better. To summarize, we can say that usability and user friendliness is more important for the diagram part than for the configuration part.

Portability The system shall be accessible from any device such as desktop computers, notebooks, tablets, and smartphones. Yet, still the aim is to support collaborative process modeling, we can assume that the screen size is reasonably big. That is, responsive design is not absolutely required.

7.2 Architecture and Design

To make the application accessible from all devices, we designed it as a classic web application, where client and server communicate via a REST API. REST (REpresentational State Transfer) (Fielding, 2000) is an architectural style for building distributed systems. It emphasizes scalability, generality of interfaces, deployment independence, and intermediate components.

The server is decomposed into three layers: model, data access, and controller. The *model* layer is the central layer. It represents the conceptual model of the domain, i.e. process models, in Java. The *data access* layer manages the mapping between the model and its representation in the database. The *controller* layer exposes the model via a REST API.

Conceptual Model Figure 7.4 shows how the `Model` entity is composed of other entities in a diagram. Figure 7.5 divides these other entities into configuration and flow part, and further shows all the relationships between them. Both diagrams are given in crow's foot notation, a form of entity-relationship diagrams.

The configuration part basically consists of the entity types `Attribute`, `Phase`, `Craft`, and `Activity`. Two additional entity types `AttributeHierarchyLevel` and `AttributeHierarchyNode` are used to model the building structure of a phase in a hierarchical way. An `AttributeHierarchy-Level` connects a phase to an attribute and indicates the position of the attribute in the hierarchy of the phase. For example, to model the skeleton phase with a hierarchy of two attributes *sector* and *level*, one would create two records of `AttributeHierarchyLevel` that reference the same phase, one with position 1 referencing the `Attribute` *sector* and one with position 2 referencing the `Attribute` *level*. Thus, each `AttributeHierarchyLevel` represents one level of granularity in the hierarchy of a building. Given a phase, all `AttributeHierarchyLevel`s connected to that phase describe the way we structured the building.

To capture the actual set of CUs in a given structure, we use the entity type `Attribute-HierarchyNode`. Each `AttributeHierarchyNode` has a value, an `AttributeHierarchyLevel` and an optional parent `AttributeHierarchyNode`. The `AttributeHierarchyLevel` connects the node to some `Phase` and to the `Attribute` which the value specifies. At first sight, it might seem that this is already sufficient to model the hierarchy, and that the parent relationship is not required. However, we need it to restrict the possible combinations of values for the given attributes. For

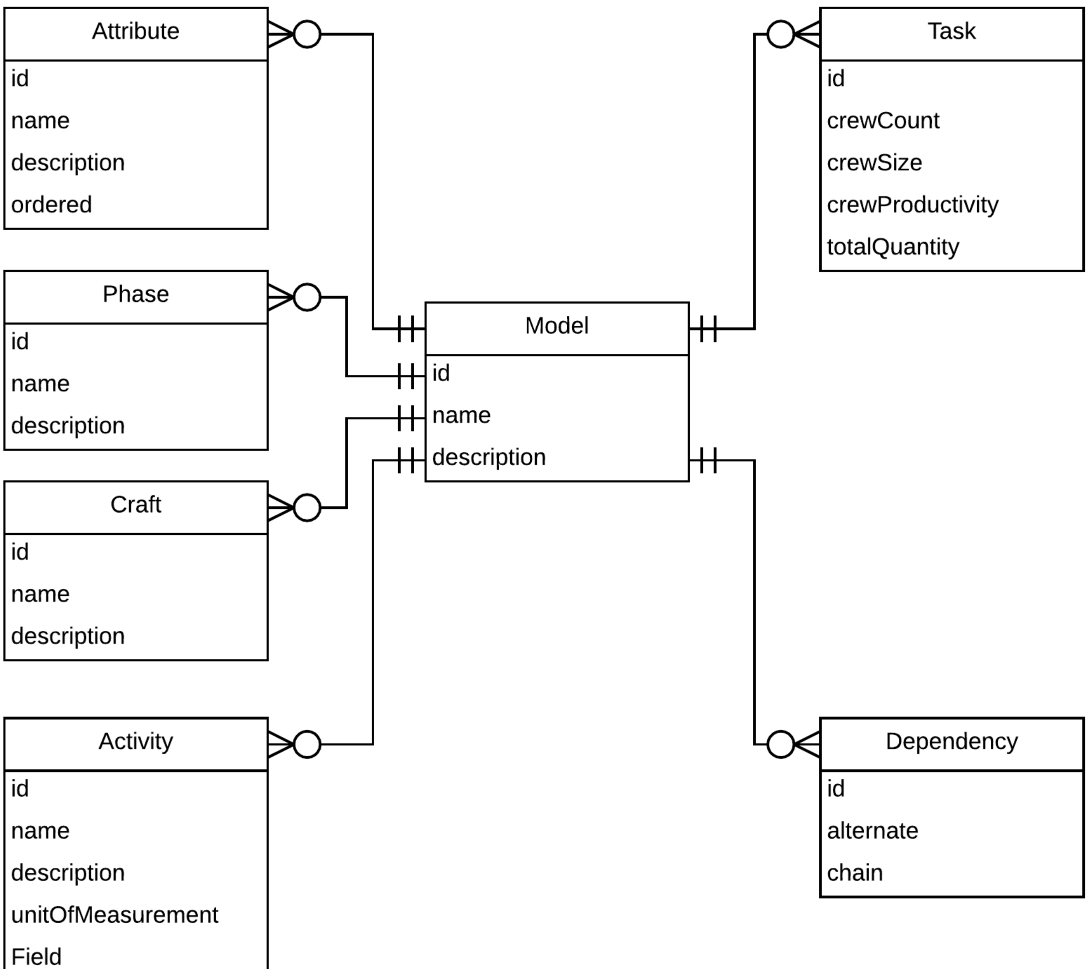


Figure 7.4: Overall conceptual model.

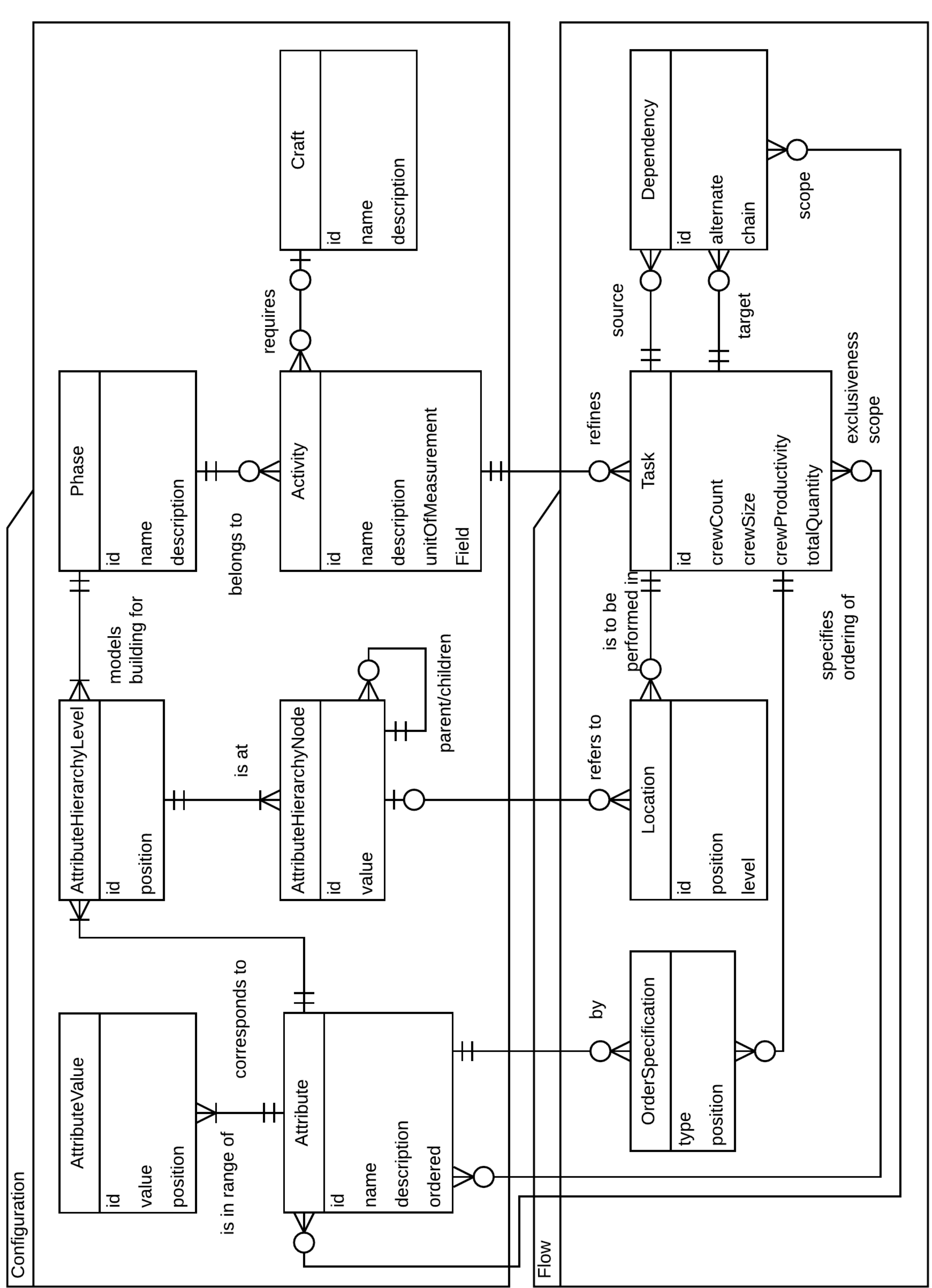


Figure 7.5: Conceptual model of a process model.

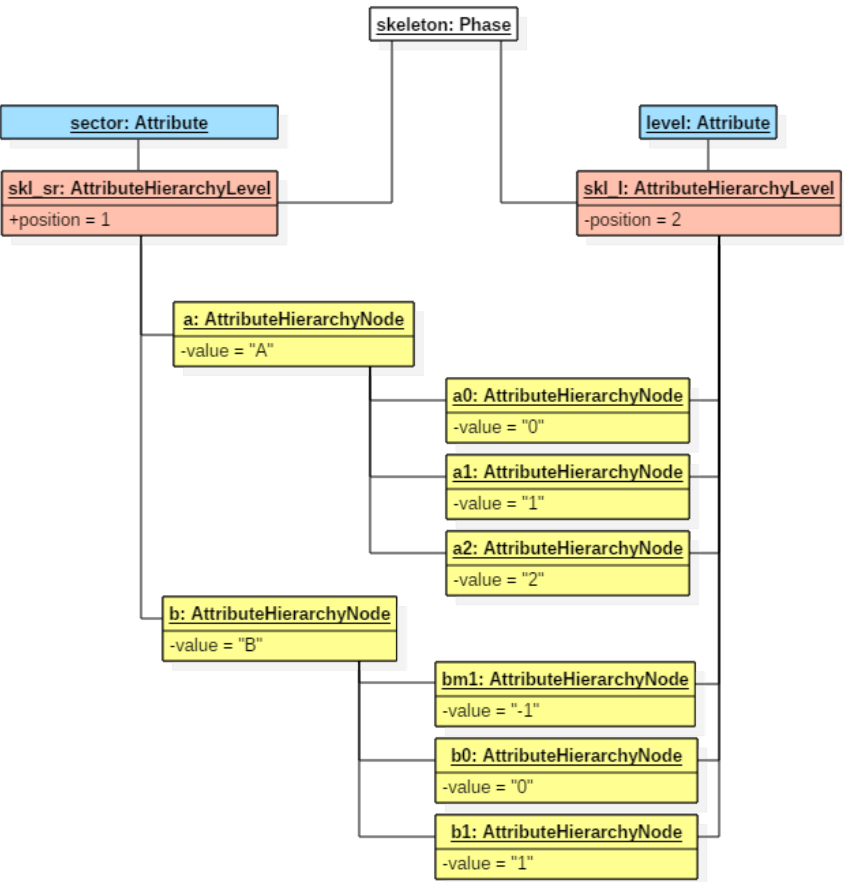


Figure 7.6: UML object diagram showing a building structure for the skeleton phase. Colors are used to highlight the different entity types.

example, consider the following CU relation Cu_{skl} for the skeleton phase:

$$Cu_{skl} \subseteq sector \times level$$
$$Cu_{skl} = \{\langle A, 0\rangle, \langle A, 1\rangle, \langle A, 2\rangle, \langle B, -1\rangle, \langle B, 0\rangle, \langle B, 1\rangle\}$$

While the levels 0 and 1 exist in both sectors, level 2 only exists in sector A and level -1 only exists in sector B. To capture this in the hierarchy, we need eight `AttributeHierarchyNodes` as shown in Fig. 7.6: one for each sector, and one for each level in each sector.

GUI Design The client is composed of several screens, where each of them is able to handle different use cases. The entry point of the application is a screen that shows the list of models currently stored in the system. A screenshot is shown in Fig. 7.7. Here, the user can create, delete, import, and export models. Figure 7.8 shows the various options for exporting files. Note that there is a section of advanced options not mentioned in the requirements. The first option allows to export a model as a NuSMV module, i.e. a transition system and an LTL formula corresponding to the model in the format expected by the NuSMV model checker. The two other options allow to export a JSON representation of the disjunctive TU graph corresponding to a model, and an orientation of it, if one exists. These advanced options are mainly useful for testing purposes.

Further, one can change the name and description of a model. For editing the more details of a model, the user needs to *open* it, which brings him to a new screen. If the opened model is empty, the configuration screen is shown, where the configuration part can be edited. It consists of three sub-views: building, crafts, and activities. The building sub-view is shown in Fig. 7.9. It lets the user select a phase and view its details. Editing options are limited here due to time constraints during the development. Only the color of a phase can be changed. To change the building structure, or to create or delete phases, one needs to import a new configuration. A

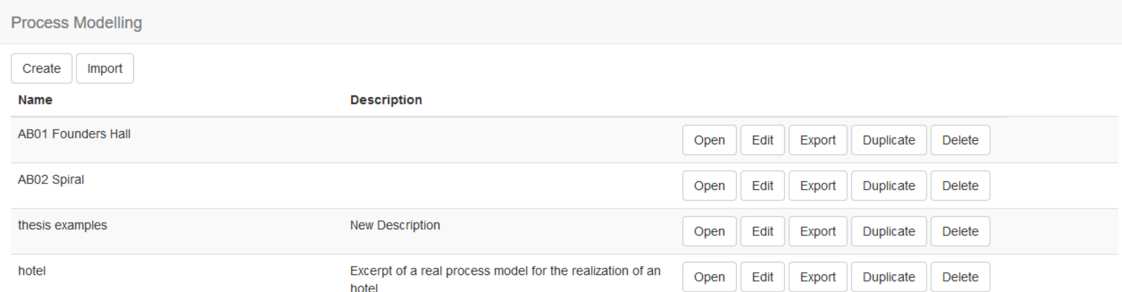


Figure 7.7: List of all models.

configuration can be imported either from an existing one in the system or from a local file. Ideally, also the editing of the building structure and the deletion of a phase should be supported by the interface. However, due to time limits during the implementation, we decided to give priority to the implementation of the functionalities concerning the definition of a model.

The craft and activities sub-views provide a list of the currently stored objects of the respective type and allow the user to create, edit, and delete them. However, deletion might fail if the object is already used in the diagram.

When a configuration exists, the user can edit the diagram in the diagram screen. There are four available sections in the top-left corner, each providing different tools:

File Import or export only the flow part of the model (see Fig. 7.10).

Edit Add or delete tasks and dependencies (see Fig. 7.11).

View Hide details such as locations and dependency scopes by unchecking the corresponding checkbox (Fig. 7.12).

Search Ask the system to highlight tasks that match some criteria. Two modes are supported for specifying criteria: a simple mode and an advanced mode. In the simple mode, the user just enters some text that, for every task, is matched against the ID, the name of the activity, the name of the craft, and the name of the phase. In the advanced mode, the user can specify a value for each of these fields to be matched

Independently of the currently selected section, the user can open a side panel to view and edit all the details of a task or a dependency by clicking on the the corresponding shape. It can be seen in Fig. 7.11.

Further, at the bottom-left corner, the application indicates whether the currently displayed diagram is consistent or not. If the user clicks on the indication, a list of problems is opened. As shown in Fig. 7.14, clicking on a problem highlights the corresponding elements in the diagram. Here, we distinguish between *structure warnings*, *structure errors*, and *consistency errors*. A structure warning is a hint that the diagram should be rewritten in a better way, e.g. because there are overlapping locations, but if this is not done, it is not a crucial problem. Structure errors, instead, indicate that there is missing something in the diagram. For example, a task without a phase, or a dependency that is not connected to two tasks, leads to a structure error. Only consistency errors indicate that the diagram is indeed inconsistent. An implementation of Algorithm 3 is used to find consistency errors. If there are any consistency errors, The application reports whether the diagram is inconsistent by checking whether there are any consistency errors.

Textual Representation For the textual representation, we decided to use a JSON syntax, because it is both easy to parse and human-readable. Also, because JSON is a standard syntax, there are software tools available for parsing JSON documents in various programming language, so we do not need to write a parser ourselves. The JSON representation of a process model is saved in files that have the extension `.mdl`. We therefore call this representation MDL.

Listing 7.1 shows the main structure of MDL files. They contain a JSON object with properties `model`, `configuration`, and `diagram`. The `model` is an object containing a single property

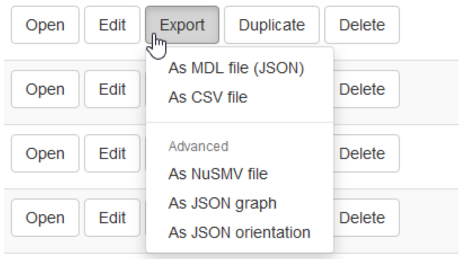


Figure 7.8: Export options.

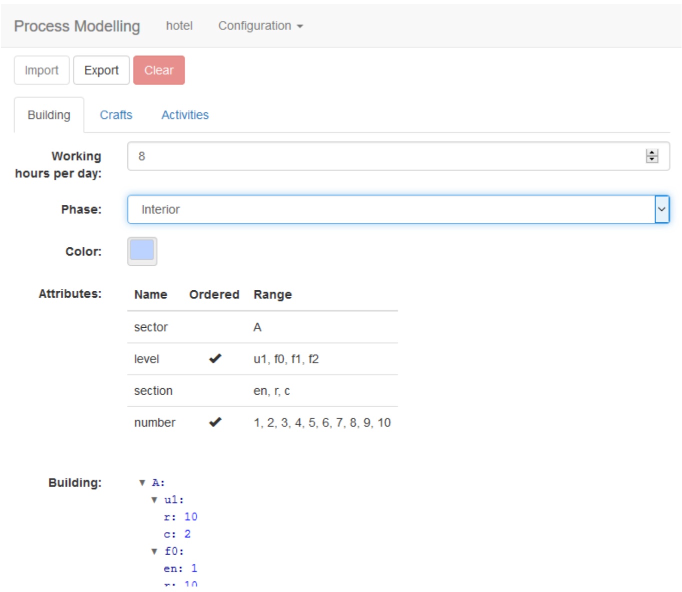


Figure 7.9: Configuration of phases and building structure.

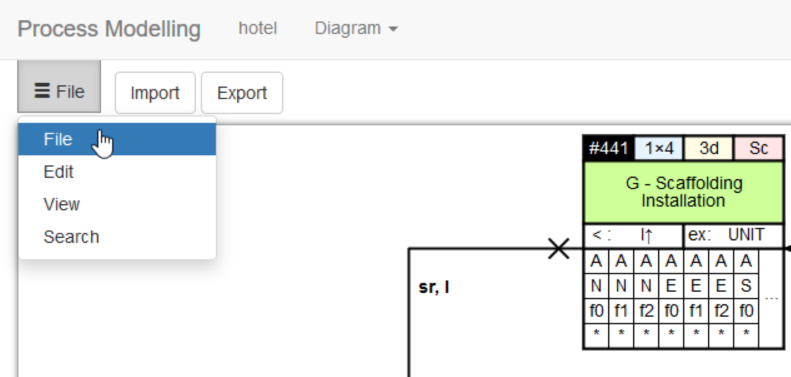


Figure 7.10: GUI for importing or exporting the flow part.

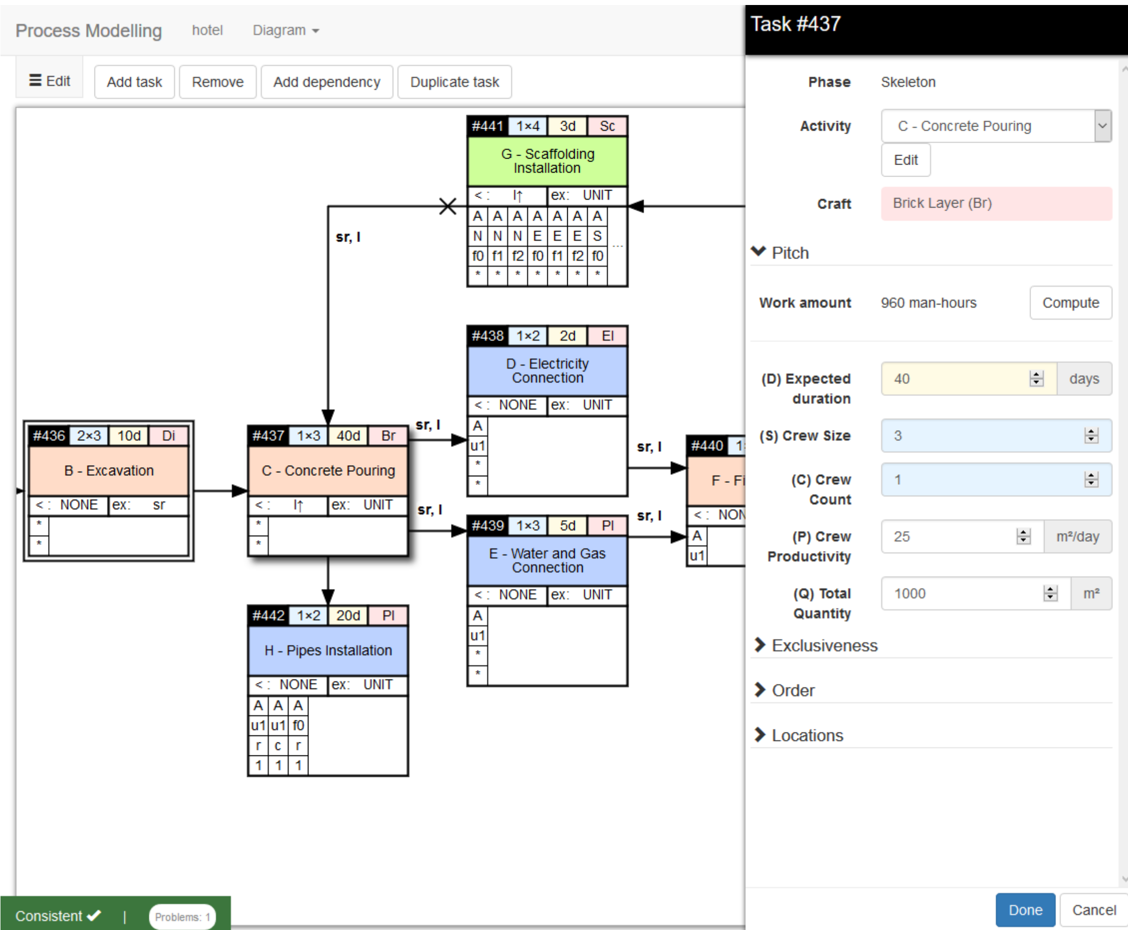


Figure 7.11: GUI for editing a diagram, e.g. the properties of a task.

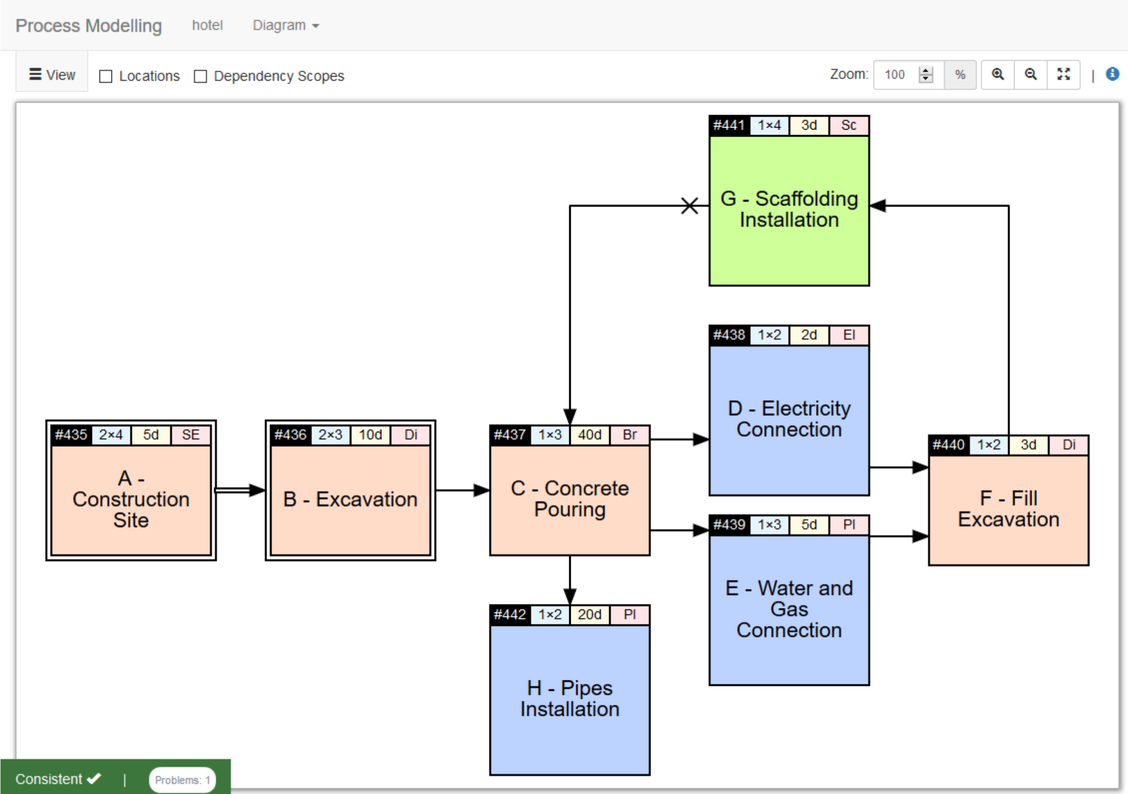


Figure 7.12: Simplified view of a diagram.

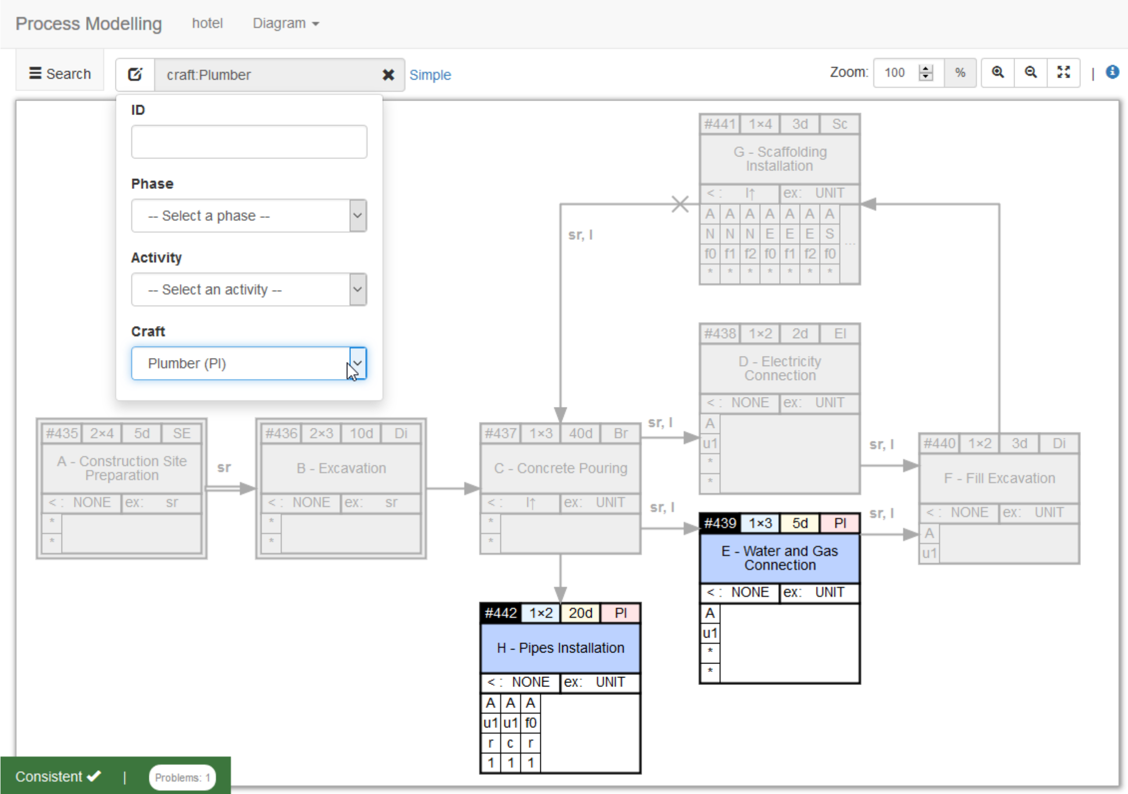


Figure 7.13: Advanced search functionality.

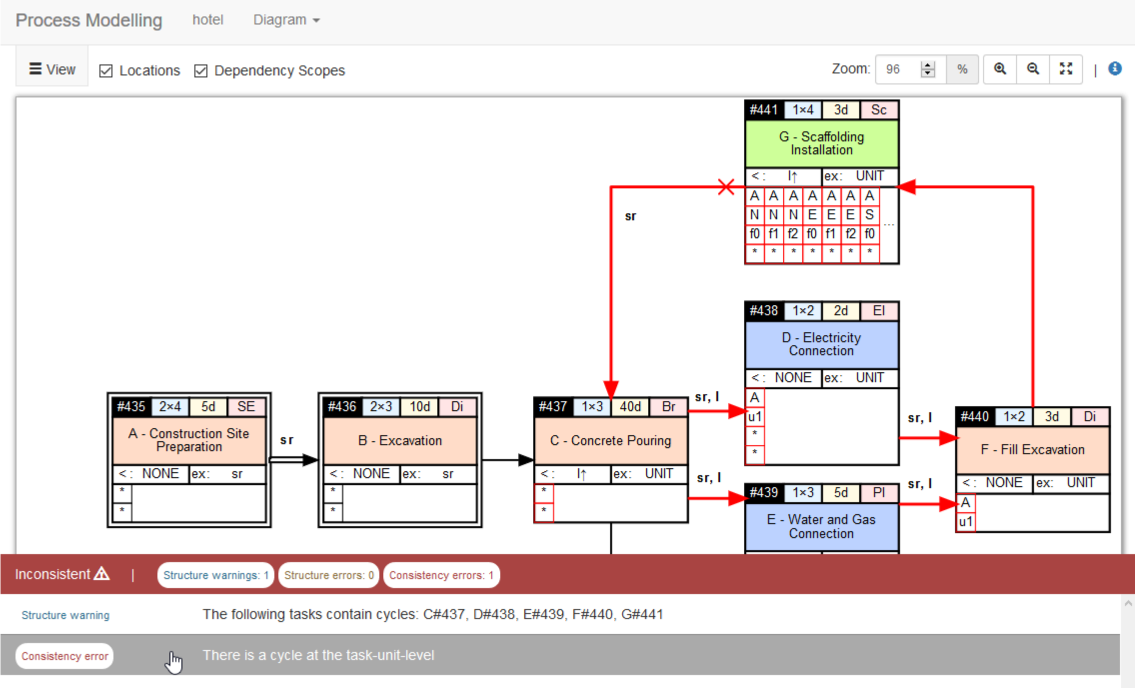


Figure 7.14: List of problems and highlighting involved shapes in the diagram.

Listing 7.1: Structure of an .mdl file

```
{
  "model" : {
    "description" : "Structure of an .mdl file"
  },
  "configuration" : {
    "hoursPerDay" : 8,
    "attributes" : [ ... ],
    "phases": [ ... ],
    "crafts" : [ ... ],
    "activities" : [ ... ]
  },
  "diagram" : {
    "tasks" : [ ... ],
    "dependencies" : [ ... ]
  }
}
```

Listing 7.2: Example of a phase in an .mdl file

```
{
  "name" : "Interior",
  "color" : { "r" : 188, "g" : 210, "b" : 255 },
  "attributes" : [ "sector", "level", "section", "number" ],
  "valueTree" : {
    "A" : {
      "u1" : {
        "r" : 10,
        "c" : 2
      },
      "f0" : {
        "en" : 1,
        "r" : 10,
        "c" : 2
      },
      "f1" : {
        "r" : 10,
        "c" : 1
      },
      "f2" : {
        "r" : 10,
        "c" : 1
      }
    }
  }
}
```

`description`. The name of a model instead is given by the filename. The `configuration` property describes the configuration part of a model. Therefore, it contains the working `hoursPerDay`, `attributes`, `phases`, `crafts`, and `activities`. The `diagram` property describes the flow part and thus contains `tasks` and `dependencies`.

Attributes, crafts, and activities all have a `name` and a `shortName`. The `name` is the preferred way of referring to these entities. Also within MDL files, we use the `name` for cross-references, e.g. to specify the craft of an activity.

Phases, instead, do not have a short name, because they are not displayed in the diagram. Listing 7.2 shows an example of a phase in the MDL representation. It contains a `name`, a `color`, and the building structure. The example models a CU relation

$$Cu \subseteq sector \times level \times section \times number$$

To indicate this, there is a list of `attributes`. The `valueTree` defines the concrete CUs on these attributes in a nested JSON structure. The property names of the outer-most JSON object of in the `valueTree` correspond to the values of first attribute. Each of these values is mapped to the CUs that it contains by another nested JSON object, starting with the next attribute.

In the example, the first attribute is *sector*, so the outer-most property names specify the different values of *sector* in this phase. Here, there is only a single sector A. In the `valueTree`, A is mapped to another JSON object, whose property names indicate the different levels that are containd in sector A, because *level* is the next attribute in `attributes`. The last attribute, which is *number*, is described by a single integer number n instead of a JSON object. This is a shortcut

Listing 7.3: Example of a task in an .mdl file

```
{
  "id" : 3,
  "activity" : "Concrete Pouring",
  "pitch" : {
    "crewSize" : 3,
    "crewCount" : 1,
    "durationDays" : 40,
    "quantityPerDay" : 25.0,
    "totalQuantity" : 1000,
  },
  "exclusiveness" : {
    "type" : "UNIT",
    "attributes" : [ "sector", "level" ]
  },
  "order" : [ {
    "attribute" : "sector",
    "orderType" : "PARALLEL"
  }, {
    "attribute" : "level",
    "orderType" : "ASCENDING"
  } ],
  "position" : {
    "x" : 220.0,
    "y" : 210.0
  },
  "locations" : [ {
    "level" : "*",
    "sector" : "*"
  } ]
}
```

Listing 7.4: Example of a dependency in an .mdl file

```
{
  "source" : 1,
  "target" : 2,
  "alternate" : false,
  "chain" : false,
  "scope" : {
    "type" : "ATTRIBUTES",
    "attributes" : [ "level" ]
  }
}
```

to refer to all integers from 1 to n, which only makes sense for the values of the last attribute. Otherwise, if one needs to specify other sets of values for the last attribute, the values can be listed in a JSON array. The same two options exist for specifying the range of an attribute.

Listing 7.3 shows the MDL representation of a task. An `id` identifies the task within the file. The `activity` of the task is specified by name. The pitch parameters are nested under the `pitch` property in their own JSON object.

The scope of the `exclusiveness` constraint of a task is given as an object with two properties: `type` and `attributes`. The `type` indicates whether this is a `GLOBAL` scope, a `UNIT` scope, or a general scope containing a set of attributes. If the latter is the case, the `type` is set to `ATTRIBUTES`, and the names of the corresponding attributes are listed in the `attributes` property.

The ordering constraint on the task is specified by the `order` property. It has a list of objects corresponding to order expressions. Each of these objects has a `orderType` property specifying the ordering operator by the literals `ASCENDING`, `DESCENDING`, or `PARALLEL`, as well as an `attribute` property specifying the attribute the operator is applied to.

Finally, the CUs of the task are given in the `locations` property. It is a list of objects corresponding to the columns in the CU table of a task in the diagram. Each column is a mapping from the attributes to the value of the corresponding cell. If an attribute is omitted or mapped to '`*`', also the diagram will contain the wild-card symbol (*) to indicate that all CUs of all values of this attribute should be considered.

Finally, Listing 7.4 shows the MDL representation of a dependency. The `source` and `target` tasks are referenced by ID. Two boolean fields `alternate` and `chain` indicate whether this depen-

dency is an alternate precedence and/or a chain precedence, respectively. If both are set to `false`, the dependency is a basic precedence. The `scope` of the dependency is specified in the same way as in the exclusiveness constraint of a task.

7.3 Implementation

In this section, we discuss which tools we used to implement the system.

Programming languages In a web application, the client runs in the web browser, where only HTML, CSS, and JavaScript can be used. In principle, there are possibilities to develop the client in other languages and translate them to JavaScript in a compile step, but that adds more complexity to the setup. Thus, we used these languages directly.

For the server, there are more possibilities. We decided to use Java, because it is object-oriented and statically typed, and has a large community with many tools and libraries available for free.

Persistence and Data Access We chose MySQL, i.e. a relational database, to be able to rely on ACID properties (Atomicity, Consistency, Isolation, Durability) and because we do not expect to accumulate Big Data. For accessing the database from Java, we use the Java Persistence API (JPA). JPA specifies a standard interface that allows to annotate Java classes with persistence metadata such as uniqueness constraints and relationships. For using JPA, one must use a persistence provided that implements JPA. We chose Hibernate, but because JPA is a standard interface, the provider as well as the underlying database can be exchanged easily at any time.

REST For implementing the controller layer, i.e. the REST API, we used Spring Data REST. Spring Data REST is a Java software module that automatically generates a REST API that exposes JPA entity classes as resources. This approach greatly reduces the amount of code required for implementing REST APIs and thereby simplifies and speeds up the development. By default, these resources are represented in JSON using the HAL (Hypertext Application Language) format, and Jackson is used to build and parse such resources.

Client structure For structuring the client JavaScript code, we use the AngularJS framework. AngularJS allows to build reusable components and a provides templating system with two-way databinding support. Further, we use RequireJS to manage dependencies among script files and to load them in the right order.

Diagram For displaying the diagram on the client, we use JointJS. JointJS is specifically designed for creating interactive graph-based diagrams on the web. It provides useful features out-of-the-box such as draggable nodes, or control-vertices on links that can be created, moved, and deleted to control the shape of links. Nodes and links are rendered as SVG elements.

7.4 Validation

To understand whether the application is fit for use in practice, it is important to understand how companies currently work in practice. Therefore, we participated in a real planning workshop at the company Frener & Reifer (F&R), which constructs and installs innovative building envelopes worldwide. The goal of the workshop was to identify the main tasks and the resulting costs for building the facade of a museum of Audemars Piguet in Lausanne. It consists of two parts: *i)* the Founder's Hall and *ii)* the Spiral. The discussion focused on one part at a time. Accordingly, we developed one process model for each part, respectively. There were several participants with different roles:

- The *project manager* is responsible of the project, and specifically for planning the project in detail. Therefore, he collected information on the project regarding tasks, their sequence, required resources etc. from the other participants.
- The *head of the technical office* designs technical solutions. In the meeting, he showed hand-drawn sketches to communicate these solutions to the other participants.

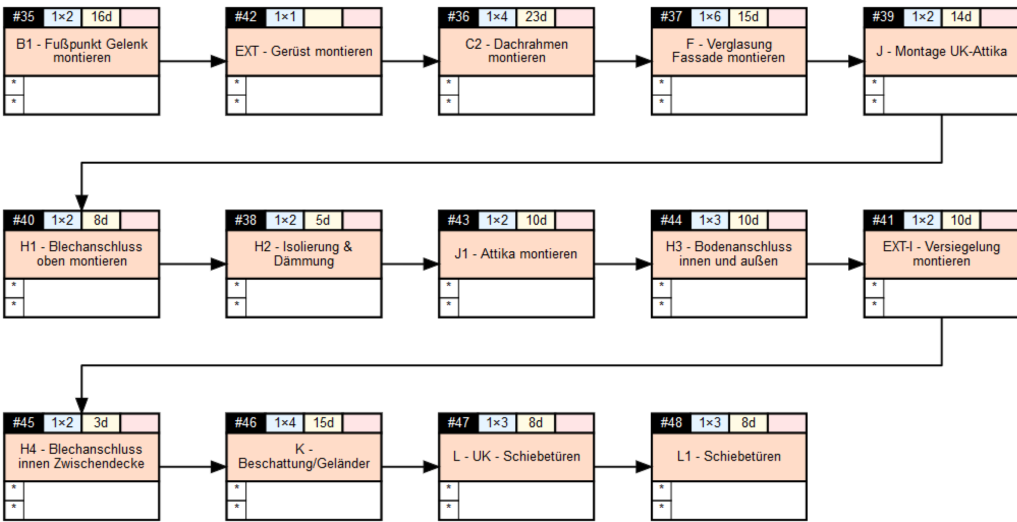


Figure 7.15: Process model diagram resulting from the planning workshop. All tasks belong to the skeleton phase. The building is structured into a single sector and a single level, resulting in only one CU.

– Another *technical office member* clarified the details of the building design by projecting 2D and 3D models from his laptop to a screen.

– The *installation foreman* used his rich experience on-site to identify the required tasks, resources, and possible difficulties for realizing the project. That is, most inputs for the discussion came from the installation foreman.

– The *moderator* documented the results of the discussion on a whiteboard as an ordered list of tasks with the expected duration and the required crew size, i.e. the pitch parameters. He guided the whole process by ensuring that all details of a task are discussed, and that tasks are discussed one after the other.

– The *secretary* compiled a MS Excel sheet based on the information written on the whiteboard. This also allowed to automatically compute the costs in terms of required man-hours based on the estimated values, both for each task and in total.

Since the application was still under development during the workshop, we used this opportunity more to check that the developed prototype and the conceptual model were adequate to satisfy the real needs of a modeling workshop, rather than to substitute it to the traditional way of performing the workshop (i.e. with whiteboards).

Specifically, we took the passive role of an additional secretary in the meeting, but entering the information into the developed web application. Figure 7.15 shows the resulting model of the Spiral part. It is basically an sequence of tasks, ordered by global basic precedences. The same is true for the process model concerning the Founder's Hall.

On the one hand, the meeting showed that the application as well as the modeling language itself are capable of representing this real world scenario, which is the most important requirement. Additionally, we learned how one wants to use the application in practice, and extended it to better support these use cases.

One observation, for instance, was that from time to time, activities are renamed. In our original prototype, this required switching from the diagram view to the configuration view and browsing through the activity table to find the one to be renamed. This, however, is not very practical to do during a workshop. Therefore, in a new version, we add the possibility of editing an activity name directly form the diagram view.

Another aspect that was improved based on the lessons learned from the meeting is the flexibility in which pitch parameters can be entered. In the tested version, one could either manually enter the estimated total duration of a task or let the application compute it based on the planned number of crews, the total quantity, and the estimated crew productivity. In practice, however, it also

happens that e.g. the total duration is estimated, and the resulting crew productivity is instead computed from the other parameters. Therefore, the improved version now allows to enter any combination of pitch parameters, and computes the one that is missing if possible. If all parameters are specified, the application checks whether they are consistent, i.e. whether computing one of them from the others results in the specified value.

Furthermore, we learned that a construction project potentially involves several process models. Currently, the application only holds a flat list of models that are considered independent of each other. Users need to manually keep track of which models are contained in the same project, e.g. by using naming conventions. For the future, we consider extending the application by the notion of a *project*, which collects several process models

Chapter 8

Conclusion and Future Work

8.1 Summary

We identified problems of ambiguity in the original PRECISE approach for process modeling and solved them by refining the language and introducing a formal semantics. On top of these, we introduced a graph-based representation that captures all constraints in a process model and showed how to use it for checking consistency. Further, we presented an application that supports defining process models in the PRECISE modeling language. The application and the underlying modeling language were tested in a planning workshop. Experiments on the implementation of the proposed algorithm indicate that the algorithm performs well in practice.

8.2 Future Work

An important aspect to investigate in the future is to understand whether the various constraints of the proposed extension of the PRECISE modeling language are both necessary and sufficient to model real world projects by applying it in multiple such projects. In particular, one should conduct more case studies in real projects, preferably involving multiple companies, to better understand their actual needs.

One assumption that we made in this thesis is that, for simplicity, the approach is targeted to repetitive projects, because quantities and productivity measures are defined on a task box level instead of a location level. Hence, we assume that all locations of a task are similar. While it is possible to define several tasks when locations differ significantly, this quickly becomes cumbersome. Therefore, one should extend the approach, e.g. allowing the total quantity and the estimated crew productivity to be specified per location per task, which is similar to how this is handled in the Location-Based Management System (LBMS).

An interesting extension of the language would be to define partitions on the locations to which alternate precedences apply. For example, when defining an alternate precedence from Scaffolding Installation to Concrete Pouring at the scope of *sector*, *level*, we might want to allow to execute these tasks in parallel for two different sectors. However, according to the proposed definition, only one CA at scope *sector*, *level* can be executed at a time. For such cases, a partitioning mechanism similar to the one supported for the definition of ordering constraints would be required.

Further, the complexity of the proposed consistency checking algorithm should be analyzed in more detail, but we could not do this in the scope of this thesis. Similarly, one should conduct more experiments on process models of real projects to better understand how the proposed consistency checking algorithm performs in practice. In this context, it would also be interesting to know whether the problem of checking consistency of a process model is NP-hard in general. If that turns out to be the case, it would mean that our algorithm, which seems to be fast enough in practice, is indeed very good, because finding an algorithm that solves the problem in polynomial time would be just as hard as for any other NP-hard problem. Furthermore, for being able to handle larger process models, one should investigate how to make the algorithm and its the implementation more memory-efficient.

Concerning the application, we focused our development efforts on those parts that are directly related to the process modeling and consistency checking approaches discussed in the theoretical chapters. Some features that are more general purpose, but nevertheless needed for being actually used by a project manager or a foreman, are left to be implemented in the future due to time constraints.

For instance, to protect the data from being viewed and change by an unauthorized person, the application should provide a login mechanism. Also, the application should support editing the configuration of a process model in the graphical interface. Currently, this is only possible by directly editing the textual representation, which is not as comfortable as a proper GUI. In practice, however, typically a process model can be configured by a single person before the workshop, so this is not a big problem.

Another aspect that could be improved is that currently, every model is considered to be independent of other models. But, as reported in Section 7.4, having the notion of a *project*, which can have multiple models, would be useful in practice. Also, it is currently not possible to define *complex* tasks, i.e. tasks which themselves are described by process models at a more detailed level. In the overall process model, this would allow to focus on interfaces between companies by using complex tasks to hide company-specific details.

More generally, this thesis focuses on process modeling, which is only one of the three phases in PRECISE which are planned to be supported by IT. Therefore, future work will focus on developing support for the other phases, and then on integrating the resulting application into one interconnected system. The interface between process modeling and scheduling is particularly interesting, because it is challenging from an algorithmic point of view but also has great potential in supporting project managers. While we discussed some basic approaches on how to check conformance of a schedule to a model, or how to generate a schedule from a model, these topics need to be investigated in more detail. In this context, one should consider combining our approach with the various graphical and algorithmic scheduling tools of LBMS.

Similarly, one should consider interoperability with BIM. In particular, it would be useful to import a BIM into the application as a starting point for the configuration part. Not only would this greatly improve the current process of configuring process models, but it would also be possible to link CAs to a 3D model of the building, so users can easily understand what CAs actually refer to.

Moreover, one strength of the proposed process modeling language is that it is very flexible in the sense that no particular building structure is assumed, but one can define a custom structure using custom attributes. In principle, this allows to use the language in contexts other than construction projects. We have seen one example in Theorem 7, where a single attribute is used to represent machines that can process only one operation at a time. To better understand the power of the proposed language, one should consider applying it to more scenarios of different domains.

Similarly, for checking consistency of a process model, we introduced the disjunctive TU graph representation. This representation is also very general, and thus applicable to various domains, at least in theory. Therefore, one should investigate whether the disjunctive TU graph representation is also a useful representation for other problems. If this is the case, then also the proposed algorithms are available for solving those other problems.

Bibliography

W.M.P. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In M. Bravetti, M. Nunez, and G. Zavattaro, editors, *International Conference on Web Services and Formal Methods (WS-FM 2006)*, volume 4184 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, Berlin, 2006.

Tamer F Abdelmaguid. Permutation-induced acyclic networks for the job shop scheduling problem. *Applied Mathematical Modelling*, 33(3):1560–1572, 2009.

Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008. ISBN 978-0-262-02649-9.

Herman Glenn Ballard. *The last planner system of production control.* PhD thesis, The University of Birmingham, 2000.

Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

Peter Brucker, Bernd Jurisch, and Bernd Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, 49(1):107 – 127, 1994. ISSN 0166-218X. doi: http://dx.doi.org/10.1016/0166-218X(94)90204-6. URL http://www.sciencedirect.com/science/article/pii/0166218X94902046.

Jacek Błażewicz, Wolfgang Domschke, and Erwin Pesch. The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research*, 93(1): 1 – 33, 1996. ISSN 0377-2217. doi: http://dx.doi.org/10.1016/0377-2217(95)00362-2. URL http://www.sciencedirect.com/science/article/pii/0377221795003622.

Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*, pages 359–364. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-45657-5. doi: 10.1007/3-540-45657-0_29. URL http://dx.doi.org/10.1007/3-540-45657-0_29.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, 2001. ISBN 978-0-262-03270-4. URL http://books.google.de/books?id=Nmc4wEaLXFEC.

P. Dallasega, D.T. Matt, and Daniel Krause. Design of the Building Execution Process in SME Construction Networks . In *2nd International Workshop on Design in Civil and Environmental Engineering*, 2013.

P. Dallasega, E. Marengo, W. Nutt, L. Rescic, D.T. Matt, and E. Rauch. Design of a Framework for Supporting the Execution-Management of Small and Medium sized Projects in the AEC-Industry. In *4th International Workshop on Design in Civil and Environmental Engineering*, 2015.

Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000.

Philippe Fortemps and Maciej Hapke. On the disjunctive graph for project scheduling. *Foundations of Computing and Decision Sciences*, 22(3):195–209, 1997.

Michael R Garey, David S Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

James E Kelley Jr and Morgan R Walker. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 160–173. ACM, 1959.

Russell Kenley and Olli Seppänen. *Location-based management for construction: Planning, scheduling and control*. Routledge, 2006.

Russell Kenley and Olli Seppänen. Location-based management of construction projects: part of a new typology for project scheduling methodologies. In *Winter Simulation Conference*, pages 2563–2570. Winter Simulation Conference, 2009.

Dominik Matt, Cristina Benedetti, Daniel Krause, and Irene Paradisi. build4future-interdisciplinary design: From the concept through production to the construction site. In *Proceedings of the 1st International Workshop on Design in Civil and Environmental Engineering*, pages 52–62, 2011.

National Building Information Model Standard Project Committee and others. Frequently Asked Questions About the National BIM Standard-United States^{TM}. URL `https://www.nationalbimstandard.org/faqs`. [Online; accessed 16-January-2017].

Object Management Group, Inc. Business Process Model and Notation (BPMN) Version 2.0. Technical report, January 2011. URL `http://taval.de/publications/BPMN20`.

Out-Law. Building Information Modelling, April 2012. [Online; accessed 16-January-2017].

Maja Pesic and Wil MP Van der Aalst. A declarative approach for flexible business processes management. In *International Conference on Business Process Management*, pages 169–180. Springer, 2006.

Olli Seppanen, Russell Kenley, et al. Performance measurement using location-based status data. In *13th International Group for Lean Construction Conference: Proceedings*, page 263. International Group on Lean Construction, 2005.

Olli Seppänen, Glenn Ballard, and Sakari Pesonen. The combination of last planner system and location-based management system. *Lean Construction Journal*, 6(1):43–54, 2010.

Olli Seppänen, Ralf-Uwe Modrich, and Glenn Ballard. Integration of last planner system and location-based management system. 2015.

A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32 (3):733–749, July 1985. ISSN 0004-5411. doi: 10.1145/3828.3837. URL `http://doi.acm.org/10.1145/3828.3837`.

Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2): 146–160, 1972.

Peter JM Van Laarhoven, Emile HL Aarts, and Jan Karel Lenstra. Job shop scheduling by simulated annealing. *Operations research*, 40(1):113–125, 1992.

Inc Vico Software. Vico Office Suite. URL `http://www.vicosoftware.com/products/Vico-Office`. [Online; accessed 16-January-2017].