Construction Process Modeling: Representing Activities, Items and their Interplay

Elisa Marengo*, Werner Nutt*, Matthias Perktold+

Faculty of Computer Science
Free University of Bozen-Bolzano, Italy
*firstname.lastname@unibz.it
+matthias.perktold@hotmail.com

Abstract. General purpose process modeling approaches are meant to be applicable to a wide range of domains. To achieve this result, their constructs need to be general, thus failing in capturing the peculiarities of a particular application domain. One aspect usually neglected is the representation of the items on which activities are to be executed. As a consequence, the model is an approximation of the real process, limiting its reliability and usefulness in particular domains.
We extend and formalize an existing declarative specification for process modeling mainly conceived for the construction domain. In our approach we model the activities and the items on which the activities are performed, and consider both of them in the specification of the flow of execution. We provide a formal semantics in terms of LTL over finite traces which paves the way for the development of automatic reasoning. In this respect, we investigate process model satisfiability and develop an effective algorithm to check it.

Keywords: Multi-instance Process Modeling · Satisfiability Checking of a Process Model · Construction Processes

1 Introduction

Process modeling has been widely investigated in the literature, resulting in approaches such as BPMN, Petri Nets, activity diagrams and data centric approaches. Among the known shortcomings of these approaches: *i)* they need to be general in order to accommodate a variety of domains, inevitably failing in capturing all the peculiarities of a specific application domain [3,9,12]; *ii)* they predominantly focus on one aspect between control flow and data, neglecting the interplay between the two [5]; *iii)* process instances are considered in isolation, disregarding possible interactions among them [1,16].

As a result, a process model is just an abstraction of a real process, limiting its applicability and usefulness in some application domains. This is particularly the case in application domains characterized by *multiple-instance* and *item-dependent* processes. We identify as *multiple-instance* those processes where several process instances may run in parallel on different items, but their execution cannot be considered in isolation, for instance because there are synchronization points among the instances or because there are limited resources for the process execution. With *item-dependent* we identify

those processes where activities are executed several times but on different items (that potentially differ from activity to activity) and items are different one from the other.

The need of properly addressing multiple-instance and item-dependent processes emerged clearly in the context of some research projects [4,6] in the construction domain. In this context, a process model serves as a synchronization mean and coordination agreement between the different companies simultaneously present on-site. Besides defining the activities to be executed and the dependencies among them, aspects that can be expressed by most of the existing modeling languages, there is the need of specifying for each activity the items on which it has to be executed, where items in the construction domain correspond to locations. In this sense, processes are *item-dependent*. Processes are also *multi-instance* since high parallelism in executing the activities is possible but there is the need to synchronize their execution on the items (e.g, to regulate the execution of two activities in the same location). The aim is not only to model these aspects, but also to provide (automatic) tools to support the modeling and the execution. This requires a model to rely on a formal semantics.

In this paper we address the problem of *multi-instance* and *item-dependent* process modeling, specifically how to specify the items on which activities are to be executed and how the control flow can be refined to account for them. Rather than defining "yet another language", we start from an existing informal language that has been defined in collaboration with construction companies and tested on-site [4,19] in a construction project (Fig. 1). We refined it by taking inspiration from Declare [2], and propose a formal semantics grounded on Linear Temporal Logic (LTL) where formulas are evaluated over finite traces [8].

Concerning the development of automatic tools, we propose an algorithm for *satisfiability* checking, defined as the problem of checking whether, given a process model, there is at least one execution satisfying it. Satisfiability is a prerequisite for the development of further automatic reasoning, such as the generation of (optimized) executions compliant to the model. We also developed a web-based prototype acting as a proof-of-concept for graphical process modeling and satisfiability checking [21]. The developed language and technique can be generalized to other application domains such as manufacturing-as-a-service [17], infrastructure, ship building and to multi-instance domains [16] such as transport/logistic, health care, security, and energy.

The paper presents the related work in Section 2, a formalization for process models and for the modeling language in Section 3, an excerpt of a real construction process model [7] in Section 4, the satisfiability problem in Section 5, and our algorithm and an implementation to check it in Section 5.2.

2 Related Work

The role of processes in construction is to coordinate the work of a number of companies simultaneously present on-site, that have to perform different kinds of work (activities) in shared locations (items). Coordination should be such that workers do not obstruct each other and such that the prerequisites for a crew to perform its work are all satisfied when it has to start. For example, in the construction of a hotel it should be possible to express that wooden and aluminum windows must be installed respectively

in the rooms and in the bathrooms, and that in the rooms the floor must be installed before the windows (not to damage them).

The adoption of IT-tools in construction is lower compared to other industries, such as manufacturing [15]. The traditional and most adopted techniques are the Critical Path Method (CPM) and the Program Evaluation and Review Technique (PERT). They consider the activities as work to be performed, focusing on their duration and the overall duration of a schedule. However, they do not account for the locations where to execute the activities and the location-based relationships between them [14]. As a result, a process representation abstracts from important details causing [22]: *i)* the communication among the companies to be sloppy, possibly resulting in different interpretations of a model; *ii)* difficulties in managing the variance in the schedule and resources; *iii)* imprecise activity duration estimates (based on lags and float in CPM and on probability in PERT); *iv)* inaccurate duration estimates not depending on the quantity of work to be performed in a location and not accounting for the expected productivity there. As a result, project schedules are defined to satisfy customer or contractual requirements, but are rarely used during the execution for process control [14].

Gantt charts are a graphical tool for scheduling in project management. Being graphical they are intuitive and naturally support the visualization of task duration and precedences among them. However, a Gantt chart already represents a commitment to one particular schedule, without necessarily relying on a process model. A process model explicitly captures the requirements for the allowed executions, thus supporting a more flexible approach: in case of delay or unforeseen events any re-schedule which satisfies the model is a possible one. Besides this limitation, Gantt charts are general-purpose and when applied to construction fail in naturally representing locations (which consequently are also not supported by IT-tools such as Microsoft Project) and have a limited representation of the precedences (only constraining two tasks, rather than, for instance, specifying constraints for tasks by floor or by room and so on). Flow-line diagrams are also a visual approach for process schedules, which explicitly represent locations and production rates. However, they also do not rely on an explicit process model. More recently, BIM-based tools have been developed. They are powerful but also require a big effort and dedicated resources to use the tool and align a BIM model with the construction site [10]. These aspects limit their use by small/medium sized companies.

From the business process literature one finds the standard BPMN. Its notation supports the representation of multi-instance activities and of data objects. However, the connection between the control flow and the data object is under-specified [5]: items and their role in ruling the control flow are not expressed explicitly. Other approaches [1,3] consider both the flow and the data and the process instances can synchronize via the *data objects*. The Instance Spanning Constraints [16] approach considers multiple instances and the constraints among them. To the best of our knowledge, none of these approaches has been applied to *execution* processes in construction (note that [3] has been applied to construction but not to the execution process, which requires to account for higher level of details). Their adoption would require adaptations and extensions, such as representation of the items and considering them in the execution flow. Similar considerations hold for Declare [2], although it supports a variety of constraint types.

Fig. 1. Process Modeling Workshop with Construction Companies and the Resulting Model.

In the context of a research [4] and a construction project [19], a new approach for a detailed modeling and management of construction processes was developed in collaboration with a company responsible for facade construction. An ad-hoc and informal modeling language was defined, with the aim of specifying the synchronization of the companies on-site and used as a starting point for the daily scheduling of the activities. To this aim, both activities and locations had to be represented explicitly. The process model of the construction project, depicted in Figure 1, was defined collaboratively by the companies participating in the project and was sketched on whiteboards. The resulting process allowed the companies to discuss in advance potential problems and to more efficiently schedule the work. The benefit was estimated in a 8% saving of the man hours originally planned and some synchronization problems (e.g., in the use of the shared crane) were discovered in advance and corresponding delays were avoided.

By applying the approach in the construction project some requirements emerged, mainly related to the ambiguity of the language (which required additional knowledge and disambiguations provided as annotations in natural language). The main requirements were: *i)* besides the activities to be performed, also the locations where to execute them need to be represented in a *structured* and *consistent* way; *ii)* to capture the desired synchronization, the specification of the flow of execution must be refined, so that not only activities are considered, but also the locations. For instance, when defining a precedence constraint between two activities it must be clear whether it means that the first must be finished everywhere before the second can start, or whether the precedence applies at a particular location i.e. floor, room and so on; *iii)* the representation of more details complicates the process management, which would benefit from the development of (automatic) IT supporting tools. In the next section we provide a formalization for process modeling in construction, paving the way for automatic tool development.

3 Multi-Instance and Item-Dependent Process Modeling

Our formalism foresees two components for a process model: a *configuration* part, defining the possible activities and items, and a *flow* part, specifying which activity is executed on which item, and the constraints on the execution. We illustrate these parts for the construction domain, although the formalization is domain-independent. As an example, we use an excerpt of a real process for a hotel construction [7].

3.1 Process Model

In this section we describe the two components of a process model.

Configuration part. The configuration part defines a set of *activities* (e.g., excavation, lay floor) and the *items* on which activities can be performed (e.g., locations).

For the item representation, we foresee a hierarchical structure where the elements of the hierarchy are the *attributes* and for each of them we define a range of possible values. The attributes to consider depend on the domain. In construction a possible hierarchy to represent the locations is depicted in Figure 2, and the attributes are: *i) sector* (sr), which represents an area of the construction site such as a separate building (as possible values we consider B1 and B2); *ii) level* (l), with values underground (u1), zero (f0) and one (f1); *iii) section* (sn), which specifies the technological content of an area (as possible values we consider room r, bathroom b, corridor c and entrance e); and *iv) unit* (u), which enumerates locations of similar type (we use it to enumerate the hotel rooms from one to four). Other attributes could be considered, such as *wall*, with values north, south, east and west, to identify the walls within a section (which is important when modeling activities such as cabling and piping). The domain values of an attribute can be ordered, for instance to represent an ascending order on the levels (u1<f0<f1). We call *item structure* a hierarchy of attributes representing an item.

In process models, e.g. in manufacturing and construction, it is common to conceptually divide a process into *phases*, where the activities to be performed and the item structure may be different. Common phases in construction are the *skeleton* and *interior*, requiring respectively a coarser representation of the locations and a more fine-grained. Accordingly, in Figure 2, an item for the skeleton phase is described in terms of sector and level only, with sector B1 having three levels and sector B2 only two. To express this, we define for each phase the *item structure*, as a tuple of attributes, and a set of *item values* defining the allowed values for the items. Accordingly, the item structure for the skeleton phase is $\langle \textit{sector}, \textit{level} \rangle$ and the possible values for sector $B1$ are $\langle B1, \text{u1} \rangle$,$\langle B1, \text{f0} \rangle$,$\langle B1, \text{f1} \rangle$, while for $B2$ they are $\langle B2, \text{u1} \rangle$,$\langle B2, \text{f0} \rangle$. Thus, an item is a sequence of values indexed by the attributes. For the interior item values see Figure 2.

More formally, a configuration $\mathcal{C}$ is a tuple $\langle At, P \rangle$, where: *i)* At is the set of attributes each of the form $\langle \alpha, \Sigma_\alpha, \alpha\uparrow \rangle$, where α is the attribute name, Σ_α is the domain of possible values, and $\alpha\uparrow$ is a linear total order over Σ_α (for simplicity, we assume all attributes to be ordered); *ii)* P is a set of phases, each of the form $\langle Ac, \mathcal{I}_s, \mathcal{I}_v \rangle$, where Ac is a set of activities; $\mathcal{I}_s = \langle \alpha_1, ..., \alpha_n \rangle$ is the item structure for the phase and $\mathcal{I}_v \subseteq \Sigma_{\alpha_1} \times ... \times \Sigma_{\alpha_n}$ is the set of *item values*.

Flow part. Based on the activities and on the items specified in the configuration part, the flow part specifies on which items the activities must be executed, for instance to express that in all levels of each sector, 'wooden window installation' must be performed in all rooms, while 'aluminum window installation' in the bathrooms. We call *task* an activity a on a set of items I, represented as $\langle \mathsf{a}, I \rangle$, where I is a subset of the possible item values for the activity's phase. We use $\langle \mathsf{a}, i \rangle$ for an activity on one item i.

Additionally, a process model must define ordering constraints on the execution of the activities. For instance, one may want to specify that the construction of the walls proceeds from bottom to top or that the floor must be installed before the windows. In the original language, the ordering constraints were declarative, i.e., not expressing

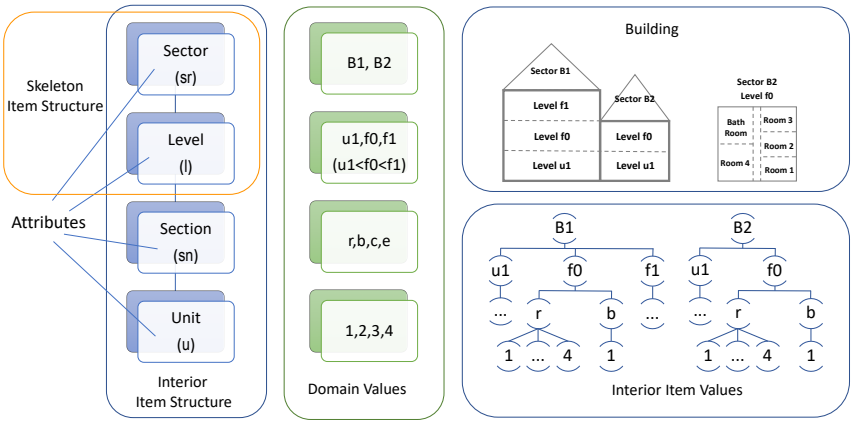

Fig. 2. Representation of the items in the hotel case study.

strict sequences but constraints to be satisfied. By representing the items on which activities are performed in a structured way, it is possible to specify the *scope* at which a precedence between two tasks applies, that is to say whether: *i)* one task must be finished on all items before progressing with the other; or *ii)* once a task is finished on an item, the other can be performed on the same item (e.g. once the floor is installed in a room, the installation of the windows can start in that room); or *iii)* the precedence applies to groups of items (e.g. once the floor is laid everywhere at one level, the windows can be installed at that level). This level of detail was one of the identified requirements (Section 2). In the original language from which this approach started, indeed, the scope of precedences was provided as disambiguation notes in natural language.

In Section 3.2 we describe our extension of the language and provide a formal semantics in terms of LTL over finite traces. Formally, a flow part $\mathcal{F}$ is a tuple $\langle T, D \rangle$, where T and D are sets of tasks and dependencies.

3.2 A Formal Language for Constraint Specification

In this section we define a language to support the definition of dependencies on task execution, i.e. the process flow. We consider a task execution to have a duration. So, for a task $\langle \mathsf{a}, i \rangle$ we represent its execution in terms of a start ($\mathsf{start}(\mathsf{a}, i)$) and an end ($\mathsf{end}(\mathsf{a}, i)$) event. An execution is then a sequence of states, each of which is defined as the set of start and end events for different tasks, that occurred simultaneously.

Our language defines a number of constructs for the specification of execution and ordering constraints on task execution. For each construct we provide an LTL formula [1] which captures the desired behavior by constraining the occurrence of the start and end events for the activities on the items. For instance, to express a precedence constraint between two tasks, the formula requires the end of the first task to occur before the start of the second. For each of them we also propose a graphical representation meant to support an overall view of the flow part. Figure 3 shows an excerpt of the flow part for

[1] In LTL, $\Box\mathsf{a}$, $\Diamond\mathsf{a}$ and $\bigcirc\mathsf{a}$ mean that condition a must be satisfied *i) always*, *ii) eventually* in the future, and *iii)* in the *next* state. Formula $\mathsf{a}\ \mathsf{U}\ \mathsf{b}$ requires condition a to be true until b.

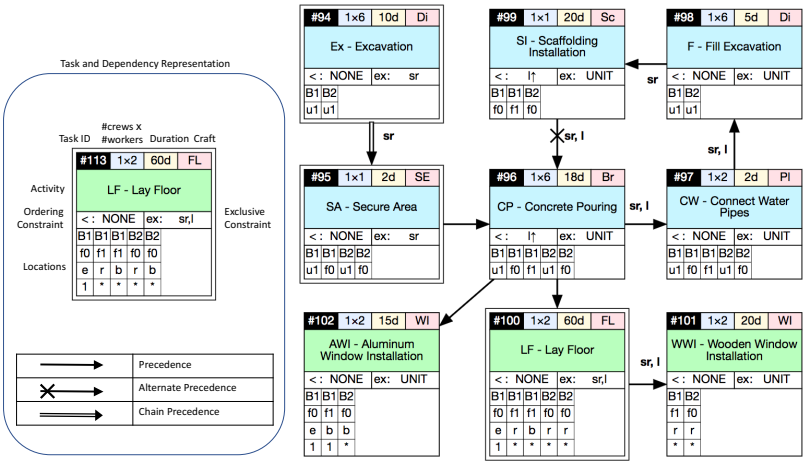

Fig. 3. Excerpt of the process model for the hotel case study. (The * denotes all possible values for the attribute).

our motivating scenario. Intuitively, each *box* is a task where the *color* represents the phase, the *label* the activity and the bottom *matrix* the items. The columns of the matrix are the item values and the rows are the attributes of the item structure. Arrows between the boxes represent binary dependencies among the tasks and can be of different kinds. The language and its graphical representation are inspired by Declare [2].

Execute. As described previously, the flow part specifies a set of tasks which need to be executed. This is captured by specifying an execute dependency *executes*(t), for each task of the flow part. It is graphically represented by drawing a box and formally an *executes*(t) constraint for $\mathtt{t} = \langle \mathsf{a}, I \rangle$ is defined as:

$$\forall i \in I \quad \Diamond\, \mathsf{start}(\mathsf{a}, i)$$

Ordered execution. In some cases it is necessary to specify an order on the execution of an activity on a set of items, for instance, to express that the concrete must be poured from the bottom level to the top one. To express this requirement we define the *ordered execution* construct having the form *ordered_execution*$(\langle \mathsf{a}, I \rangle, \mathcal{O})$. This constraint specifies that the activity 'a' must be executed on all items in I following the order specified in $\mathcal{O}$. We express $\mathcal{O}$ as a tuple of the form $\langle \alpha_1 o_1, ..., \alpha_m o_m \rangle$ where α_i is an attribute and o_i is an *ordering operator* among $\uparrow$ or $\downarrow$. The expression $\alpha_i\uparrow$ refers to the linear total order of the domain of α (defined in the configuration), while $\alpha_i\downarrow$ refers to its inverse. Given the set of items I, these are ordered according to $\mathcal{O}$ in the following way. The items are partitioned so that the items with the same value for α_1 are in the same partition set. The resulting partition sets are ordered according to $\alpha_1 o_1$. Then, each partition set is further partitioned according to α_2 and each resulting partition set is ordered according to $\alpha_2 o_2$, and so on for the remaining operators. This iterative way of partitioning and ordering defines the ordering relation $<_{\mathcal{O},I}$, based on which precedence constraints are defined to order the execution of the activity a on the items.

As an example, consider the task Concrete Pouring (CP) in Figure 3. To specify that it must be performed from bottom to top, we graphically use the label *<:l↑*, which

corresponds to the constraint *ordered_execution*($\langle CP, I\rangle$,l↑), meaning that the items I are partitioned according to their values for the level (regardless of the sector), and then ordered. As a result, the activity must be performed at level u1 before progressing to f0 (and then f1). Formally, a constraint *ordered_execution*($\langle \mathsf{a}, I\rangle, \mathcal{O}$) is:

$$\textit{executes}(\langle \mathsf{a}, I\rangle) \quad \text{and} \quad \forall \langle i_1, i_2\rangle \in <_{\mathcal{O},I} \ \textit{precedes}(\langle \mathsf{a}, \{i_1\}\rangle, \langle \mathsf{a}, \{i_2\}\rangle)$$

Precedes (auxiliary construct). The formula above relies on the *precedes*($\langle \mathsf{a}, I_\mathsf{a}\rangle, \langle \mathsf{b}, I_\mathsf{b}\rangle$), auxiliary construct which requires an activity a to be executed on a set of items I_a before an activity b (potentially the same) is performed on any item in I_b. Formally:

$$\forall i_\mathsf{a} \in I_\mathsf{a}, i_\mathsf{b} \in I_\mathsf{b} \quad \neg \mathsf{start}(\mathsf{b}, i_\mathsf{b}) \ \mathsf{U} \ \mathsf{end}(\mathsf{a}, i_\mathsf{a})$$

Not interrupt (auxiliary construct). Another requirement is the possibility to express that the execution of an activity on a set of items is not interrupted by other activities. For instance, to express that once the lay floor activity starts at one level in one sector, no other task can be performed at the same level and sector, we have to express that the execution of the task on a group of items must not be interrupted, and that we group and compare the items by considering their values for sector and level only (abstracting from section and unit). To this aim, we introduce the auxiliary construct *not_interrupt*(T_1, T_2) which applies to sets of tasks T_1 and T_2 and specifies that the two sets of tasks cannot interrupt each other: either all tasks in T_1 are performed before the tasks in T_2 or the other way around (we consider sets of tasks because this will be useful later in the definition of the alternate precedence constraint). Formally, *not_interrupt*(T_1, T_2) is defined as:

$$\forall \mathsf{t}_1 \in T_1, \mathsf{t}_2 \in T_2 \quad \textit{precedes}(\mathsf{t}_1, \mathsf{t}_2) \quad \text{or} \quad \forall \mathsf{t}_1 \in T_1, \mathsf{t}_2 \in T_2 \quad \textit{precedes}(\mathsf{t}_2, \mathsf{t}_1)$$

Projection Operator. To compare two items by considering only some of the attributes of their item structure, we introduce the concept of *scope*. The scope is a sequence of attributes used to compare two items. For instance, given a scope $\mathsf{s} = \langle \text{sector}, \text{level}\rangle$, we can say that the items $\langle B1, \text{f1}, \text{room}, 1\rangle$ and $\langle B1, \text{f1}, \text{bathroom}, 1\rangle$ are equal under s. In this case, we say that the two items are at the same scope. For the comparison, we define the projection operator to project an item on the attributes in the scope.

Definition 1 (Projection Operator Π_s). *Given an item $i = \langle \mathsf{v}_1, .., \mathsf{v}_n\rangle$ and a scope $\mathsf{s} = \langle \alpha_{j_1}, .., \alpha_{j_m}\rangle$, the projection of i on* s *is $\Pi_\mathsf{s}(i) = \langle \mathsf{v}_{j_1}, .., \mathsf{v}_{j_m}\rangle$ with $\mathsf{v}_{j_h} = \alpha_{j_h}(i)$.*

This means, in particular, that for the empty scope $\mathsf{s} = \langle\rangle$, we have $\Pi_{\langle\rangle}(i) = \langle\rangle$, and thus $\forall i, i' \ \Pi_{\langle\rangle}(i) = \Pi_{\langle\rangle}(i')$. When applied to a set of items I, the result of the projection operator with scope s is the set (without duplicates), obtained by applying the projection operator to every item $i \in I$. In other words, it is the set of possible values for the attributes in s, w.r.t. the items in I.

Exclusive execution. An *exclusive execution* constraint *exclusive_execution*($\langle \mathsf{a}, I_\mathsf{a}\rangle, \mathsf{s}$) expresses that once an activity is executed on an item at scope s, no other activity can be performed on items at the same scope. Formally, the task has to be *executed* and for every other task having an item at scope s, the two tasks must not interrupt each other.

For a scope $\mathsf{s} = \langle \alpha_{j_1}, .., \alpha_{j_m} \rangle$, let $\pi_\mathsf{s} = \langle \mathsf{v}_{j_1}, .., \mathsf{v}_{j_m} \rangle$ be a tuple of values for the attributes in s. We use the selector operator $\sigma_{\pi_\mathsf{s}}(I)$ to select the items in I having the values specified in π_s for the attributes s . Formally, *exclusive_execution*$(\langle \mathsf{a}, I_\mathsf{a} \rangle, \mathsf{s})$ is:

$$\begin{array}{c} \textit{executes}(\langle \mathsf{a}, I_\mathsf{a} \rangle) \quad \text{and} \quad \forall \langle \mathsf{b}, I_\mathsf{b} \rangle \in T \text{ and } \langle \mathsf{b}, I_\mathsf{b} \rangle \neq \langle \mathsf{a}, I_\mathsf{a} \rangle, \\ \forall \pi_\mathsf{s} \in \mathit{\Pi}_\mathsf{s}(I_\mathsf{a}), \forall i_\mathsf{b} \in \sigma_{\pi_\mathsf{s}}(I_\mathsf{b}) \,^2 \quad \textit{not\_interrupt}(\{\langle \mathsf{a}, \sigma_{\pi_\mathsf{s}}(I_\mathsf{a}) \}\rangle, \{\langle \mathsf{b}, \{i_\mathsf{b}\}\rangle\}) \end{array}$$

As a special case, when $\mathsf{s} = \langle\rangle$ the execution of the entire task cannot be interrupted. By default, tasks have an exclusive constraint at the finest-granularity level for the items, i.e. two activities cannot be executed at the same time on the same item.

An exclusive execute constraint (except the default at the item scope) is represented with a double border box and the scope is specified in the slot labeled with *ex*. In Figure 3, lay floor has an exclusive execution constraint *ex*:(sr,l) for sector and level.

We now introduce *binary* dependencies that specify ordering constraints between pairs of tasks. By representing also the items, we can specify precedences at different scopes: *i)* task (a task must be finished on all items before the second task can start); *ii)* item scope (once the first task is finished on an item, the second task can start on the item); *iii)* between items at the same scope (e.g. a task must be performed in all locations of a floor before another task can start on the same floor). This is visualized by annotating a binary dependency (an arrow) with the sequence of attributes representing the scope. When no label is provided, the task scope is meant. In Figure 3, the dependency between concrete pouring and lay floor is at task level, while the one between lay floor and wooden window installation is labeled with sr,l, to represent the scope $\langle \text{sector}, \text{level} \rangle$: given a sector and a level, the activity lay floor must be done in every section and unit before wooden windows installation can start in that sector at that level.

Precedence. A *precedence* dependency *precedence*$(\langle \mathsf{a}, I_\mathsf{a} \rangle, \langle \mathsf{b}, I_\mathsf{b} \rangle, \mathsf{s})$ expresses that an activity a must be performed on a set of items I_a before an activity b starts on items I_b. The scope s defines whether this applies at the task, item, or item group.

$$\forall \pi_\mathsf{s} \in \mathit{\Pi}_\mathsf{s}(I_\mathsf{a}) \cap \mathit{\Pi}_\mathsf{s}(I_\mathsf{b}) \quad \textit{precedes}(\langle \mathsf{a}, \sigma_{\pi_\mathsf{s}}(I_\mathsf{a}) \rangle, \langle \mathsf{b}, \sigma_{\pi_\mathsf{s}}(I_\mathsf{b}) \rangle)$$

The formula above expresses that for the items at the same scope in I_a and I_b, activity a must be executed there before activity b. If $\mathsf{s} = \langle\rangle$ activity a must be performed on all its items before activity b can start (task scope).

Alternate precedence. Let us consider the example of the scaffolding installation and the concrete pouring: once the scaffolding is installed at one level, the concrete must be poured at that level before the scaffolding can be installed to the next level. This alternation is captured by the dependency *alternate*$(\langle \mathsf{a}, I_\mathsf{a} \rangle, \langle \mathsf{b}, I_\mathsf{b} \rangle, \mathsf{s})$, which is in the first place a precedence constraint between a and b. It also requires that once a is started on a group of items at a scope $(\sigma_{\pi_\mathsf{s}}(I_\mathsf{a}))$, then b must be performed on its items at the same scope $(\sigma_{\pi_\mathsf{s}}(I_\mathsf{b}))$, before a can progress on items at a different scope $(\sigma_{\pi'_\mathsf{s}}(I_\mathsf{a}))$:

[2] The result of $\mathit{\Pi}_\mathsf{s}(I_\mathsf{a})$ is the set of possible values for the attributes in s considering I_a. For each of them we select the items in I_b that are at the same scope, and we apply the *not_interrupt*.

$$\textit{precedence}(\langle \mathsf{a}, I_\mathsf{a}\rangle, \langle \mathsf{b}, I_\mathsf{b}\rangle, \mathsf{s}) \quad \text{and} \quad \forall \pi_\mathsf{s}, \pi'_\mathsf{s} \in \Pi_\mathsf{s}(I_\mathsf{a}) \cap \Pi_\mathsf{s}(I_\mathsf{b}), \pi_\mathsf{s} \neq \pi'_\mathsf{s}$$ [3]
$$\textit{not\_interrupt}(\{\langle \mathsf{a}, \sigma_{\pi_\mathsf{s}}(I_\mathsf{a})\rangle, \langle \mathsf{b}, \sigma_{\pi_\mathsf{s}}(I_\mathsf{b})\rangle\}, \{\langle \mathsf{a}, \sigma_{\pi'_\mathsf{s}}(I_\mathsf{a})\rangle, \langle \mathsf{b}, \sigma_{\pi'_\mathsf{s}}(I_\mathsf{b})\rangle\})$$

Graphically, an alternate precedence is represented as an arrow (to capture the precedence), and an X as a source symbol, to capture that the source task cannot progress freely, but it has to wait for the target task to be completed on items at the same scope.

Chain precedence. Finally, let us consider the case in which the execution of two tasks must not be interrupted by other tasks on items at the same scope. For instance, the tasks excavation and secure area must be performed one after the other and no other tasks can be performed in between for each sector. Note that the dependency types defined before declaratively specify an order on the execution of two tasks but do not prevent other tasks to be performed in-between. To forbid this we define the *chain precedence* dependency $\textit{chain}(\langle \mathsf{a}, I_\mathsf{a}\rangle, \langle \mathsf{b}, I_\mathsf{b}\rangle, \mathsf{s})$, which, as the alternate precedence, builds on top of a precedence dependency. Additionally, it requires that the execution of the two tasks on items at the same scope is not interrupted by other tasks executing on items at the same scope. The formula considers all tasks different from $\mathsf{t}_1 = \langle \mathsf{a}, I_\mathsf{a}\rangle$ and $\mathsf{t}_2 = \langle \mathsf{b}, I_\mathsf{b}\rangle$ sharing items at the same scope. For this it specifies a *not_interrupt* constraint. Formally,

$$\textit{precedence}(\langle \mathsf{a}, I_\mathsf{a}\rangle, \langle \mathsf{b}, I_\mathsf{b}\rangle, \mathsf{s}) \quad \text{and} \quad \forall \pi_\mathsf{s} \in \Pi_\mathsf{s}(I_\mathsf{a}) \cap \Pi_\mathsf{s}(I_\mathsf{b}),$$
$$\forall\, \mathsf{t}_3 = \langle \mathsf{c}, I_\mathsf{c}\rangle \in T, \quad \text{s.t. } \mathsf{t}_3 \neq \mathsf{t}_1 \quad \text{and} \quad \mathsf{t}_3 \neq \mathsf{t}_2$$
$$\forall\, i \in \sigma_{\pi_\mathsf{s}}(I_\mathsf{c}) \;\; \textit{not\_interrupt}(\{\langle \mathsf{c}, \{i\}\rangle\}, \{\langle \mathsf{a}, \sigma_{\pi_\mathsf{s}}(I_\mathsf{a})\rangle, \langle \mathsf{b}, \sigma_{\pi_\mathsf{s}}(I_\mathsf{b})\rangle\})$$

Graphically, it is represented as a double border arrow.

4 Process Modeling for the Hotel Scenario

The model of the hotel case study consists of roughly fifty tasks. Figure 3 reports an excerpt showing some activities of the skeleton (blue) and interior phases (green). The item structure for the skeleton phase is defined in terms of sector sr and level l, while the interior consists of sector, level, section sn, and unit number u (see Figure 2).

The task EXCAVATION belongs to the skeleton phase and must be performed on sectors $B1$ and $B2$, in both cases at the underground level u1. The activity SECURE AREA must also be performed in both sectors, but at the underground and ground floor f0. For security reasons, once the excavation is finished in one sector, the area must be secured for that sector before any other task can start. This is expressed with a *chain precedence* dependency at scope sr (sector), graphically represented as double bordered arrow. Additionally, while performing the activities, their execution in a sector cannot be interrupted by other activities. This is expressed with an *exclusive execution* at scope sector (*ex:* sr) for both tasks, represented as double border box.

Only after the area has been secured everywhere, the CONCRETE POURING can start. This is expressed with a precedence at the task scope (graphically an arrow without label). The concrete can be poured proceeding from the bottom to the top floor, captured by an *ordered constraint* ($<:\mathsf{l}\uparrow$). Note that this task has an exclusive constraint

[3] The projection operator is applied to I_a and I_b and only projections π_s and π'_s that are in common are considered. For every distinct π_s and π'_s either a and b are performed on items at scope π_s without being interrupted by executing a and b on items at scope π'_s, or vice versa.

ex: UNIT, i.e. an exclusive constraint at the finest granularity level, which expresses that two activities cannot be performed simultaneously on the same item. For all tasks the same exclusive execution at the item scope is assumed and it is not graphically highlighted. It is represented with a double border box when a coarser scope is specified.

Once the concrete has been poured at the underground level of one sector, the pipes for the water can be connected before the excavation is filled, and after this the task SCAFFOLDING INSTALLATION can start, proceeding from the bottom to the top. Once the scaffolding is installed at one level, the concrete must be poured at that level, in order to be able to install the scaffolding for the next level. This requirement is expressed by the *alternate precedence* at scope sr, l (graphically, with an arrow starting with an X).

When the concrete pouring task is finished everywhere, the lay floor task can start (which belongs to the interior phase). This task must be performed before the installation of the wooden windows, which is foreseen in all rooms, so not to damage them. This is captured by the precedence constraint between LAY FLOOR and WOODEN WINDOW INSTALLATION at scope sr, l. To be more efficient, the lay floor task has an exclusive execution constraint at scope sr, l. Since aluminum windows are less delicate than wooden ones, their installation does not depend on the lay floor task.

To support the graphical definition of a process flow, we implemented a web-based prototype [21] (see the master's thesis [20]), which we used to produce Figure 3. The prototype and the graphical language support the overall view of a process model.

5 Satisfiability Checking

When developing a tool requiring inputs from the user, one cannot assume that the provided input will be meaningful. Specifically, one cannot assume that a given model is *satisfiable*, that is there exists at least one execution satisfying it. Relying on LTL semantics would allow us to perform the check using model checking techniques. However, as reported in the experiment description (Section 5.2) this takes more than 2 minutes for a satisfiable model with 8 tasks and 9 dependencies. In this section, we describe a more effective algorithm and its performance evaluation by means of experiments.

5.1 How to Check for Satisfiability

An execution is a sequence of states defined in terms of start and end events.

Definition 2 (Execution). *Let E be the set of* start *and* end *events for the tasks of a process model. A process execution ρ of length n is a function $\rho\colon \{1, .., n\} \to 2^E$.*

There are some properties of interest that an execution is expected to satisfy in reality: *i) start-end order:* a start event is always followed by an end event; *ii) non-repetition:* an activity cannot be executed more than once on the same item, and start and end events never repeat; *iii) non-concurrence:* at most one task at a time can be performed on an item. All these properties can be expressed in LTL.[4] We say that an execution ρ is a *well-defined execution* if and only if it satisfies all of these three properties.

[4] *start-end order:* $\Box(\mathsf{start}(\mathsf{a}, i) \to \bigcirc\Diamond\mathsf{end}(\mathsf{a}, i))$;
non-repetition: $\Box(\mathsf{start}(\mathsf{a}, i) \to \bigcirc\Box\neg\mathsf{start}(\mathsf{a}, i)) \wedge \Box(\mathsf{end}(\mathsf{a}, i) \to \bigcirc\Box\neg\mathsf{end}(\mathsf{a}, i))$;
non-concurrence: $\Box(\mathsf{start}(\mathsf{a}, i) \to \neg\mathsf{start}(\mathsf{a}', i)\ \mathsf{U}\ \mathsf{end}(\mathsf{a}, i))$, where $\mathsf{a} \neq \mathsf{a}'$.

Definition 3 (Possible Execution). *A well-defined execution ρ is a* possible execution *for a process model if and only if*

i) all events occurring in ρ are of activities and items of the configuration part;
ii) for all tasks $\langle \mathsf{a}, I \rangle$ in the flow part, the task $\langle \mathsf{a}, i \rangle$ is executed in ρ for all $i \in I$;
iii) ρ satisfies all the ordering constraints specified in the flow part.

The observation underlying the algorithm that checks statisfiability is that, since all our dependencies relate the end of a task with the start of another one (end-to-start), it holds that if there is a possible execution, then there is one where all activities on an item are atomic, that is, each start event is immediately followed by the corresponding end event. Moreover, if there is such an execution, then there even exists a sequential one where no atomic activities take place at the same time. Therefore, it is sufficient to check for the existence of sequential executions of atomic activities.

The algorithm relies on an auxiliary structure that we call *activity-item* (AI) graph. Intuitively, in an AI-graph we represent each activity to be performed on an item as a node $\langle \mathsf{a}, i \rangle$, conceptually representing the execution of a on i. Ordering constraints are then represented as arcs in the graph. This allows us to characterize the satisfiability of a model by the absence of loops in the corresponding AI graph.

Let us first consider *P-models*, which are process models with precedence and ordering constraints only. Given a P-model $\mathcal{M}$, we denote the corresponding AI-graph as $\mathcal{G}_{\mathcal{M}} = \langle V, A \rangle$, where for each task $\mathsf{t} = \langle \mathsf{a}, I \rangle$ in the flow part and for each item $i \in I$ there is an AI node $\langle \mathsf{a}, i \rangle \in V$, without duplicates; for each precedence and ordering constraint we introduce a number of arcs in A among AI nodes in V. For instance, a precedence constraint between two tasks at the task scope is translated into a set of arcs, linking each AI node corresponding to the source task to each AI node corresponding to the target task. A precedence constraint at the item scope is translated into arcs between AI nodes of the two activities on the same items.

Theorem 1. *A P-model $\mathcal{M}$ is satisfiable iff the graph $\mathcal{G}_{\mathcal{M}}$ is cycle-free.*

Proof (Idea). If $\mathcal{G}_{\mathcal{M}}$ does not contain cycles, the nodes can be topologically ordered and the order is a well-defined execution satisfying all ordering constraints in $\mathcal{M}$. A cycle in $\mathcal{G}_{\mathcal{M}}$ corresponds to a mutual precedence between two AI nodes, which is unsatisfiable.

We now consider general models, called *G-models*, where all types of dependency are allowed. First, let us consider an exclusive constraint *exclusive_execution*$(\langle \mathsf{a}, I_{\mathsf{a}} \rangle, \mathsf{s})$. It requires for each scope $\pi_{\mathsf{s}} \in \Pi_{\mathsf{s}}(I_{\mathsf{a}})$, that the execution of a on the items in the set $\sigma_{\pi_{\mathsf{s}}}(I_{\mathsf{a}})$ is not interrupted by the execution of other activities at the same scope. Considering an activity b to be executed on an item i_{b} at the same scope (i.e., $\Pi_{\mathsf{s}}(i_{\mathsf{b}}) = \pi_{\mathsf{s}}$), the exclusive constraint is not violated if the execution of b occurs *before or after* the execution of a on all items in $\sigma_{\pi_{\mathsf{s}}}(I_{\mathsf{a}})$. We call *exclusive group* a group of AI nodes, whose execution must not be interrupted by another node. We connect this node and the exclusive group with an undirected edge, since the execution of the node is allowed either before or after the exclusive group. Then, we look for an orientation of this edge, such that it does not conflict with other constraints, i.e., it does not introduce cycles. With chain and alternate precedences, we deal in a similar way. Indeed, both require that the execution of two tasks on a set of items is not interrupted by other activities (chain) or by the same activity on other items (alternate).

To represent a *not_interrupt* constraint in a graph we introduce *disjunctive activity-item* graphs (DAI-graphs) inspired by [11]. Given a *G-model* $\mathcal{M}$, the corresponding DAI-graph is $\mathcal{D}_{\mathcal{M}} = \langle V, A, X, E \rangle$, where $\langle V, A \rangle$ are defined as in the AI-graph of a P-model,[5] $X \subseteq 2^V$ is the set of exclusive groups, and $E \subseteq V \times X$ is a set of undirected edges, called *disjunctive edges*, connecting single nodes to exclusive groups. A disjunctive edge can be *oriented* either by creating arcs from the single node to each node in the exclusive group or vice versa, so that all arcs go in the same direction (i.e., either outgoing from or incoming to the single node). An *orientation* of a DAI-graph is a graph that is obtained by choosing an orientation for each edge. We say that a disjunctive graph is *orientable* if and only if there is an acyclic orientation of the graph.

Theorem 2. *A G-model $\mathcal{M}$ is satisfiable iff the DAI-graph $\mathcal{D}_{\mathcal{M}}$ is orientable.*

Proof (Idea). If $\mathcal{D}_{\mathcal{M}}$ is orientable, there exist a corresponding acyclic oriented graph. Then, there exists a well-defined execution satisfying all precedence constraints (see Theorem 1). The non_interrupt constraints are satisfied by construction of $\mathcal{D}_{\mathcal{M}}$. If $\mathcal{D}_{\mathcal{M}}$ is not orientable, a topological order satisfying all the constrains does not exist (see [20]).

5.2 An Implementation for Satisfiability Checking

Algorithm. To check for the orientability of a DAI-graph $\mathcal{D}_{\mathcal{M}} = \langle V, A, X, E \rangle$ we develop an algorithm that is based on the following observations.

Cycles. If the graph $\mathcal{G}_{\mathcal{M}} = \langle V, A \rangle$ contains a cycle, then $\mathcal{D}_{\mathcal{M}}$ is not orientable.

Simple edges. Disjunctive edges where both sides consist of a single node, called *simple edges*, can be oriented so that they do not introduce cycles (see the thesis [20]).

Resolving. Consider an undirected edge between a node u and an exclusive set of nodes $S \in X$. If there is a directed path from u to a node $v \in S$ (or the other way around), then there is only one way to orient the undirected edge between v and S without introducing cycles. We call this operation *resolving*.

Partitioning. Sometimes, one can partition a DAI-graph $\mathcal{D}_{\mathcal{M}}$ into DAI-subgraphs such that $\mathcal{D}_{\mathcal{M}}$ is orientable if and only if each of these DAI-subgraph is orientable. Then each such subgraph can be checked independently.

Let us discuss the partitioning operation in more detail. Given $\mathcal{D}_{\mathcal{M}} = \langle V, A, X, E \rangle$, we are looking for a partition of the node set V into disjoint subsets, the partition sets, satisfying two conditions: *i)* nodes of an exclusive group belong all to the same subset; *ii)* there are no cycles among the subsets, that is, by abstracting each subset to a single node and preserving the arcs connecting different subsets, there is no subset that can be reached from itself. It is possible to prove for such partitions that the original DAI-graph is orientable if and only if each partition set, considered as a DAI-graph, is orientable [20]. Intuitively, if the partition sets do not form a cycle, then they can be topologically ordered, and the AI nodes in each partition set can be executed respecting the order. Since an exclusive group is entirely contained in one partition set, execution according to a topological order also satisfies the exclusive constraint.

[5] Including also the directed arcs to represent the precedence constraints of the chain and alternate dependencies (see the formalization in Section 3.2)

To obtain such a partition, we temporarily add for each pair of nodes in an exclusive group two auxiliary arcs, connecting them in both directions. Then we compute the strongly connected components (SCCs) of the extended graph and consider each of them as a partition set. (It may be that there is only one of them.) After that we drop the auxiliary arcs. This construction ensures that *i)* an exclusive group is entirely contained in a SCC; and *ii)* there are no cycles among the partition sets because a cycle would cause the nodes to belong to the same partition set.

Below we list our boolean procedure SAT that takes as input a DAI-graph $\mathcal{D}$ and returns "true" iff $\mathcal{D}$ is orientable. It calls the subprocedure NDSAT that chooses a disjunctive edge and tries out its possible orientations.

procedure SAT($\mathcal{D}$)
  drop all simple edges in $\mathcal{D}$
  resolve all orientable disjunctive edges in $\mathcal{D}$
  if $\mathcal{D}$ contains a cycle **then**
    return false
  else partition $\mathcal{D}$, say into $\mathcal{D}_1, \ldots, \mathcal{D}_n$
    if NDSAT($\mathcal{D}_i$) = true for all $i \in \{1, \ldots, n\}$ **then**
      return true
    else return false

procedure NDSAT($\mathcal{D}$)
  if $\mathcal{D}$ has a disjunctive edge e **then**
    orient e in the two possible ways, resulting in $\mathcal{D}_+$, $\mathcal{D}_-$
    return SAT($\mathcal{D}_+$) or SAT($\mathcal{D}_-$)
  else return true

SAT itself performs only deterministic steps that simplify the input, discover unsatisfiability, or divide the original problem into independent subproblems. After that NDSAT performs the non-deterministic orientation of a disjunctive edge. Since the calls to NDSAT at the end of SAT are all independent, they can be run in parallel.

Experiments. We ran our experiments on a desktop PC with eight cores Intel i7-4770 of 3.40 GHz. We tested the performance of NuSMV, a state-of-the-art model checker, on a process model similar to the one reported in Figure 3, consisting of 8 tasks and 9 dependencies, resulting in a DAI-graph of 236 nodes. The NuSMV model checker with Bounded Model Checking took 2 min 35 sec on a satisfiable model. We also considered different inconsistency scenarios: *i)* a DAI-graph with a cycle *ii)* an acyclic DAI graph that is non-orientable; *iii)* a DAI-graph similar to case *ii)*, but with an increased number of nodes. On all these unsatisfiable cases, we aborted the check if exceeding 1 hour.

We implemented our algorithm in Java and tested it on the same variants of the hotel scenario on which we tested the model checker. The results are reported in Table 1 (top). As can be seen, the implementation outperforms the NuSMV model checker both in the satisfiable and unsatisfiable variants. In order to understand the performance of the implementation w.r.t. the model size, we performed some experiments by increasing the number of tasks and dependencies in the non-orientable variant of the model. We chose this variant because it is the most challenging for the algorithm, which has to find a partition and non-deterministically chose an orientation for the undirected edges.

Model	Tasks	Dependencies	Nodes	Arcs	Edges	Time (ms)
satisfiable	8	9	236	9415	524	27
cyclic	8	9	236	10003	521	5
non-orientable	12	14	244	9435	574	10
non-orientable	12	14	424	15131	1740	23
non-orientable	60	75	2,120	76,103	10,635	598
	120	173	4,240	168,681	42,470	1,189
	180	296	6,360	361,969	95,505	3,682
	300	623	10,600	674,584	265,175	14,199
	360	822	12,720	948,099	381,810	24,223
	480	1,291	16,960	1,436,759	678,680	55,866
	720	2,562	25,440	3,082,925	1,526,820	379,409
	960	4,187	33,920	5,217,426	2,714,160	OOM

Table 1. Experimental results.

The results are shown in Table 1 (bottom). On a model of 180 tasks, which we believe represents an average real case scenario, the performances are still acceptable (around 4 seconds). The implementation took around 1 minute on a model of 480 tasks, which is acceptable for an offline check. It ran out of memory (OOM) on a model of 960 tasks (too many for most of real cases). More details on the experiments are reported in the thesis [20]. The web-based prototype [21] that we developed implements the satisfiability checking and the export of the NuSMV file of a process model.

6 Conclusions and Future Work

This work presents an approach for process modeling that represents activities, items and accounts for both of them in the control flow specification. We investigate the problem of satisfiability of a model and develop an effective algorithm to check it. The algorithm has been implemented in a proof-of-concept prototype that also supports the graphical definition of a process model [21,18].

The motivation for developing a formal approach for process modeling emerged in the application of non-formal models in real projects [4,19], which resulted in improvements and cost savings in construction process execution. This opens the way for the development of automatic tools to support construction process management. In this paper we presented the statisfiability checking, starting from which we are currently investigating the automatic generation of process schedules, optimal w.r.t. some criteria of interest (e.g., costs, duration). To this aim we are investigating the adoption of constraint satisfaction and (multi-objective) optimization techniques. We are also investigating the use of Petri Nets for planning [13]. We will apply modeling and automatic scheduling to real construction projects in the context of the research project COCkPiT [6].

Acknowledgments. This work was supported by the projects MoMaPC, financed by the Free University of Bozen-Bolzano and by COCkPiT financed by the European Regional Development Fund (ERDF) Investment for Growth and Jobs Programme 2014-2020.

References

1. van der Aalst, W.M.P., Artale, A., Montali, M., Tritini, S.: Object-Centric Behavioral Constraints: Integrating Data and Declarative Process Modelling. In: Description Logics (2017)
2. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative Workflows: Balancing Between Flexibility and Support. Computer Science-R&D 23(2) (2009)
3. van der Aalst, W., Stoffele, M., Wamelink, J.: Case Handling in Construction. Automation in Construction 12(3), 303–320 (2003)
4. Build4Future Project. www.fraunhofer.it/en/focus/projects/build4future.html
5. Calvanese, D., De Giacomo, G., Montali, M.: Foundations of Data-Aware Process Analysis: a Database Theory Perspective. In: PODS. ACM (2013)
6. COCkPiT: Collaborative Construction Process Management. www.cockpit-project.com/
7. Dallasega, P., Matt, D., Krause, D.: Design of the Building Execution Process in SME Construction Networks . In: 2nd International Workshop DCEE (2013)
8. De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on LTL on Finite Traces: Insensitivity to Infiniteness. In: AAAI. AAAI Press (2014)
9. Dumas, M.: From Models to Data and Back: The Journey of the BPM Discipline and the Tangled Road to BPM 2020. In: BPM. LNCS 9253, Springer (2015)
10. Forsythe, P., Sankaran, S., Biesenthal, C.: How Far Can BIM Reduce Information Asymmetry in the Australian Construction Context? Project Management Journal 46(3) (2015)
11. Fortemps, P., Hapke, M.: On the Disjunctive Graph for Project Scheduling. Foundations of Computing and Decision Sciences 22 (1997)
12. Frank, U.: Multilevel Modeling - Toward a New Paradigm of Conceptual Modeling and Information Systems Design. Business & Information Systems Engineering 6(6) (2014)
13. Hickmott, S.L., Sardiña, S.: Optimality Properties of Planning Via Petri Net Unfolding: A Formal Analysis. In: Proceedings of ICAPS (2009)
14. Kenley, R., Seppänen, O.: Location-Based Management for Construction: Planning, Scheduling and Control. Routledge (2006)
15. KPMG International: Building a Technology Advantage. Harnessing the Potential of Technology to Improve the Performance of Major Projects. Global Construction Survey (2016)
16. Leitner, M., Mangler, J., Rinderle-Ma, S.: Definition and Enactment of Instance-Spanning Process Constraints. In: Web Information Systems Engineering. LNCS 7651 (2012)
17. Lu, Y., Xu, X., Xu, J.: Development of a Hybrid Manufacturing Cloud. Journal of Manufacturing Systems 33(4) (2014)
18. Marengo, E., Dallasega, P., Montali, M., Nutt, W.: Towards a Graphical Language for Process Modelling in Construction. In: CAiSE Forum 2016. CEUR Proceedings, vol. 1612 (2016)
19. Marengo, E., Dallasega, P., Montali, M., Nutt, W., Reifer, M.: Process Management in Construction: Expansion of the Bolzano Hospital. In: Business Process Management Cases. Springer (2018)
20. Perktold, M.: Processes in Construction: Modeling and Consistency Checking. Master's thesis, Free University of Bozen-Bolzano (2017), http://pro.unibz.it/library/thesis/00012899S_33593.pdf
21. Perktold, M., Marengo, E., Nutt, W.: Construction process modeling prototype, http://bp-construction.inf.unibz.it:8080/ConstructionProcessModelling-beta
22. Shankar, A., Varghese, K.: Evaluation of Location Based Management System in the Construction of Power Transmission and Distribution Projects. In: 30th International Symposium on Automation and Robotics in Construction and Mining (2013)