CoPModL: Construction Process Modeling Language and Satisfiability Checking.

Elisa Marengo[a,*], Werner Nutt[a], Matthias Perktold[b]

[a]*Faculty of Computer Science, Free University of Bozen-Bolzano, Italy*
name.surname@unibz.it
[b]*Faculty of Computer Science, Free University of Bozen-Bolzano, Italy*
matthias.perktold@asaon.com

Abstract

Process modeling has been widely investigated in the literature and several general purpose approaches have been introduced, addressing a variety of domains. However, generality goes to the detriment of the possibility to model details and peculiarities of a particular application domain. As acknowledged by the literature, known approaches predominantly focus on one aspect between control flow and data, thus neglecting the interplay between the two. Moreover, process instances are not considered or considered in isolation, neglecting, among other aspects, synchronization points among them. As a consequence, the model is an approximation of the real process, limiting its reliability and usefulness in particular domains. This observation emerged clearly in the context of a research project in the construction domain, where preliminary attempts to model inter-company processes show the lack of an appropriate language.

Building on a semi-formal language tested on real construction projects, in this paper we define CoPModL, a process modeling language which accounts both for activities and items on which activities are to be executed. The language supports the specification of different item-based dependencies among the activities, thus serving as a synchronization specification among several activity instances. We provide a formal semantics for the language in terms of LTL over finite traces. This 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. Known shortcomings of these approaches are, among others, that *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 [1, 2, 3]; *ii)* they predominantly focus on one aspect between control flow and data, neglecting the interplay between the two [4]; *iii)* process instances are considered in isolation, disregarding possible interactions among them [5, 6].

*Corresponding author

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 domains characterized by *multiple-instance* and *item-dependent* processes. We identify as *multiple-instance* those processes where several process instances 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 modeling activities only (as in BPMN or Petri Nets) is not enough because also the items on which activities have to be performed play a role and has to be modeled. This is because items are different to one another and part of the role of a model is indeed to define *i)* what are the items; *ii)* on which items an activity has to be executed; and *iii)* how does the execution of an activity on some items synchronizes with the execution of other activities on the same or different items.

The need of properly addressing multiple-instance and item-dependent processes emerged clearly in the context of some research projects [7, 8] in the building construction domain. A construction project is composed of phases. A common approach is to organize it into six phases [9]: *i)* conception of the project, *ii)* design, *iii)* pre-construction, *iv)* procurement, *v)* execution (construction), *vi)* post-construction. In this work we consider the *execution* stage only. A process model for the execution stage aims at defining in detail the synchronization and coordination agreement between the different companies simultaneously present on-site. Thus, 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 representing for each activity the items on which it has to be executed, where items in the construction domain correspond to locations. In this sense, construction processes are *item-dependent*: *i)* locations need to be defined in a structured way and *ii)* a model has to explicitly represent in which locations an activity has to be performed (for synchronization or organizational purposes, for instance). Note indeed that not necessarily an activity has to be performed in all locations. Processes are also *multiple-instance,* since parallelism in executing the activities is possible but there is the need to synchronize their execution on the items. For instance, only one activity at a time can be executed in a location and precedence constraints may rule the execution of the activities in one location [9].

The aim of a process model is to capture these aspects in a non-ambiguous way, both in order to collect the requirements for a construction process in such a way that all participants agree on them, discuss the details of the execution and possibly to identify and solve potential problems. Also, it is desirable to rely on (automatic) tool that can support the definition and management of such models. For instance, an appealing functionality is the automatic generation of (optimized) schedule starting from a process model. As a preliminary step, however, it would be important to verify some properties of interest on the starting model, such as its satisfiability. To achieve these results, a model must rely on a formal semantics.

In this paper we address the problem of *multiple-instance* and *item-dependent* process modeling, specifically considering the building construction domain. We tackle the problems of 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 [7, 10] in a construction project (Figure 1). We introduce CoPModL (Construction Process Modeling Language), which refines and extends the constructs of the language, and define a formal semantics grounded on Linear Temporal Logic over finite traces [11] (LTL_f). The formal semantics allows us to automatically check the satisfiability of a process model, that is checking whether an execution satisfying all the requirements exists in principle. In this paper we propose an efficient algorithm to check this property. To easily communicate with the construction companies, from which we gathered the requirements for construction process modeling, we developed a web-based proof-of-concept tool which supports a graphical definition of a model and implements the satisfiability check.

CoPModL has been defined in the context of the research project COCkPiT [8], based on the requirements identified for the building construction domain. In principle the language is general and thus can be applied to other applications domains. In fact, the need of representing locations and the need of expressing formal conditions ruling the control flow already emerged in the literature [12, 13, 14]. However, we believe that due to its peculiarities CoPModL mainly suits construction related domains (e.g., civil and infrastructure projects) and production domains (e.g., manufacturing and manufacturing-as-a-service [15]).

The paper extends the work in [16], discussing more in detail the requirements of the application domain and using them to discuss in detail existing approaches and their unsuitability for construction process modeling and satisfiability checking in this setting. Formal proofs and lemmas are introduced in this paper, as well as the description of the proof-of-concept tool that we developed to support CoPModL process modeling and satisfiability checking [17]. The efficiency of the satisfiability checking algorithm is better substantiated by means of additional experiments, showing *i)* the efficiency of the algorithm compared to a model checking tool; *ii)* that the optimizations on which the satisfiability checking algorithm relies on are indeed improving the checking performances; and *iii)* the algorithm is able to handle process models of reasonable size.

The paper is structured as follows. In Section 2 we collect the main requirements that emerged in our research projects and with the collaboration with some companies; we also discuss the unsuitability of two well known languages, namely BPMN and Declare. Section 3 presents CoPModL and its formalization and Section 4 presents an example of its application. The satisfiability checking algorithm and its evaluation are described in Section 5. Section 6 presents CoPMod, a proof-of-concept tool that is developed as a web application and that supports construction process modeling and satisfiability checking. In Section 7 we discuss the related work both from the computer science and the construction domain. Conclusions end the paper in Section 8.

2. Construction Process Requirements

Before entering into the details of our proposal, let us first analyze in Section 2.1 some characteristics of the construction domain and the requirements for process modeling emerging from them. In Section 2.2 we clarify some of the requirements by discussing a possible application of the modeling languages BPMN, a well known procedural standard, and Declare, a well known declarative approach, and we show the unsuitability of these languages for the domain.

2.1. Requirements for Construction Process Modeling

Management and modeling of construction processes for the execution stage are challenging due to some peculiarities of the domain which lead to specific requirements.

Process model elements: perspectives, activities, locations, and location-based dependencies. One of the main aim of defining a process model is to capture the desired coordination among the different companies that are involved in the project. As anticipated in the introduction, indeed, construction processes are multiple-instance, meaning that parallelism in executing the activities in the locations is possible, but coordination should ensure that workers do not obstruct each other and that the prerequisites for a crew of workers to perform its work are all satisfied when it has to start. In very small projects, coordination is defined on the fly at execution time based on the experience of the workers. However, when a few different companies are involved, it is important to define and agree on the coordination so as to avoid misunderstandings, to discover possible execution problems and to potentially optimize some parts of the process. An effective way to achieve this result is by defining a process model in a collaborative way, that is by involving the main actors taking part in the construction (such as project managers and foremen of the involved companies).

In this way, indeed, it is possible to leverage on the experience of the workers in defining the coordination rather than imposing one.

In this scenario, different expertise is brought together and it is thus important that a model is able to capture the *different perspectives* at the right level of detail and that the information that is relevant for coordination purposes can be expressed explicitly and not left as common or hidden knowledge. Different perspectives emerge mainly in the representation of the activities and of the items on which they need to be performed. For example, let us consider as a scenario an excerpt of a real project for the construction of a hotel. In this case, it happened that the company responsible for the construction of the skeleton had to synchronize with the other companies, among which the one responsible for the interior. However, when representing the locations where activities had to be performed, more levels of detail were needed for the interior activities (identifying the technological content for the area, such as "office", "swimming pool", "room" and the precise location, such as room 1 at the second floor), while for the skeleton a coarser representation was needed (that is in terms of levels). Both representations were needed in the same model, as well as a relation between them (e.g. the second floor represents the same area both for interior construction and skeleton). A model must be able to accommodate both perspectives and potentially others.

When specifying a process model one must be able to specify *what* needs to be performed in terms of activities, *where* an activity has to be executed in terms of locations (accounting for different perspectives), and also to represent *location-based dependencies* on the execution of the activities. For instance, in the hotel scenario wooden windows were installed only in the rooms while aluminum windows were installed in the other locations. Since wooden windows are delicate, the companies agreed among them that in the rooms the floor would have been put first and then the wooden windows would have been installed (so that the floor could dry without damaging the windows). When defining this precedence, it must be possible to specify also at which level the dependency applies. Does the precedence in the example mean that the floor must be put everywhere before a window can be installed, or does it rather mean that the floor must be put entirely at one level of the building before the installation of the windows can start there, or even that the precedence applies room by room. In this sense, a model must be able to capture *location-based* relations specifying at which level a dependency applies.

Adequate support for one-of-a-kind projects. It is not difficult to imagine that construction processes, as opposed to manufacturing for instance, are one of a kind both from a design and organizational point of view. Design-wise, the elements to be built as well as material and technologies to be used are likely different from project to project. From the execution point of view, this affects the coordination patterns that are related to the technological and material requirements. Organization-wise, for a construction project several companies and crafts are required on-site and also in this case they are always different. As a consequence, also the coordination among them changes from project to project, at least for what concerns preferred patterns of execution and specific needs of the companies. For these reasons, only little can be reused from project to project and most of the time a process model must be defined from scratch. However, even if the coordination patterns are always different, most of the terminology that is used, such as the activity names, the name of the locations, and the terminology for the involved crafts can potentially be reused and adapted to different projects.

Flexibility and adaptability of the model. When it comes to the execution, construction processes are subject to unpredictable events [1] and to changing requirements from the customer. The advantage of having a process model is that in these cases one can evaluate several alternative countermeasures and adaptations before implementing them, thus considering the impact of the changes in the long run, rather than adopting a solution that seems to be good at the time it is taken. However, changes are possible only if the process model supports them in reality. In theory, indeed, any process model can be changed and adapted. In

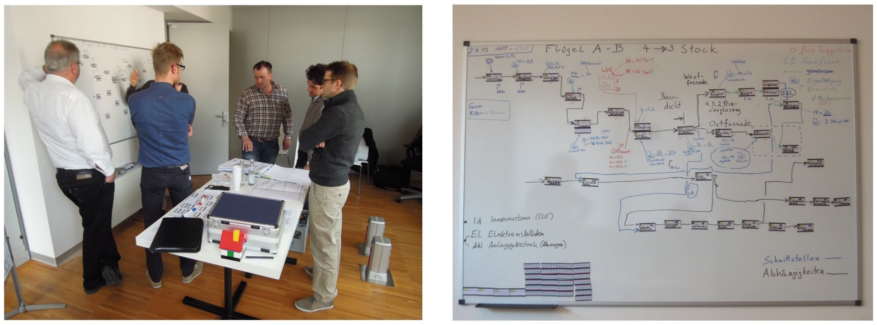


Figure 1: Process Modeling Workshop with Construction Companies and the Resulting Model.

practice, an adaptation is possible only if it is possible to understand how the model should be changed in order to obtain the desired result and if the changes to be made are circumscribed to one part of the model.

Non-ambiguity of the language. A first attempt of collaborative process modeling was part of a research [7] and a construction project [10]. Here 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 (performed manually). The process model of the construction project, depicted in Figure 1, was defined collaboratively by the companies participating in the project by means of magnetic plates representing activities and locations where to perform them, sticking them on magnetic whiteboards and drawing dependencies among them.

This approach for process modeling was evaluated positively by the involved companies because they could discuss in advance potential problems and 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. However, the language presented several ambiguities which required additional knowledge and disambiguation, often provided as annotations in natural language. Indeed, in the approach people could represent activities and locations where to perform the work, but these latter were represented in an ad-hoc and unstructured way, requiring additional knowledge to interpret them case by case and also leading to non-consistent representations of the same location in different activities. Concerning dependencies, these were all represented in the same way by means of an arrow, but they were interpreted differently depending on the case: sometimes, the meaning was that a task must be finished everywhere before another can start, other times the precedence was intended as a floor by floor constraint and so on.

Also, different *constraint types* were all represented in the same way. A loop was interpreted as two activities to be performed one after the other, *alternating* in the different floors, that is with the first one waiting for the second one before progressing on the next floor. In some cases an arrow was interpreted as a precedence (with potentially other tasks performed in between the two), while other times it was interpreted as a *strict sequence*. Finally, in some cases tasks needed an *exclusive* access to a certain area, that is, regardless of any order with the other tasks, once it is started in one area (e.g., one floor) no other tasks could access that area. This is to avoid obstructions and for efficiency reasons.

The ambiguity of the language has several drawbacks, among which the difficulties in interpreting a model for someone who is not involved in the definition process and, also very important, the impossibility to provide automatic support.

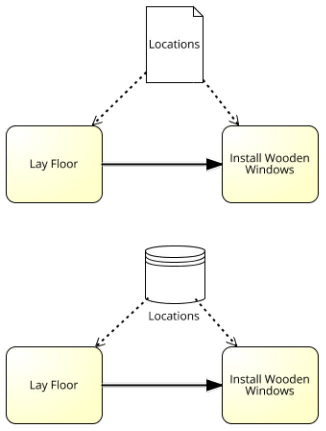
Need of IT-tools as a support. The representation of more details, like the definition of locations and location-based dependencies rather than just activities and precedences, complicates the definition of a process model, which was already perceived as a demanding activity for some workers when introduced in [10]. For instance, when introducing a new task one had to first remember where the task was foreseen and how they agreed to represent locations in the workshop (in fact, leading to inconsistencies in the representation of the locations). Also, at the end of the workshop someone had to copy the resulting process model with some tool (often inadequate for the purpose), such as Microsoft Excel and Power Point, and then share it with the other participants. These time-demanding and error-prone activities can be mitigated by providing adequate IT and automated support. After this first step, other automation steps can be undertaken, like the automatic generation of optimized schedules.

2.2. *BPMN and Declare for Construction Process Modeling*

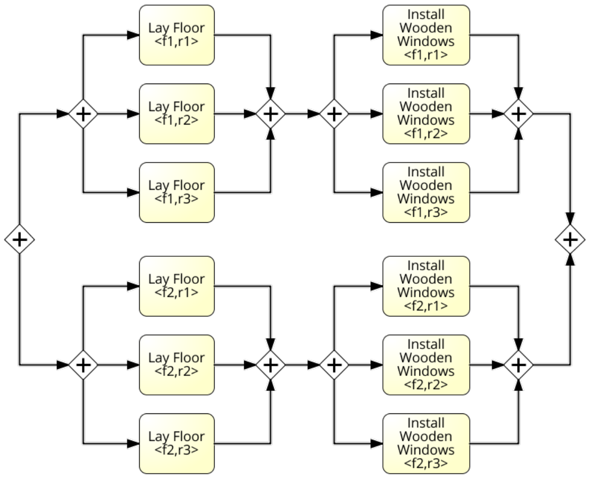
The *Business Process Modeling Notation* (BPMN) standard is a widely known approach for modeling business processes. BPMN is a graphical language which supports the definition of, among others, activities to be performed, roles performing them, events, parallel and alternative branches [18]. The language has a rich notation and the fact of being graphical supports its comprehension even by non-experts.

One of the recognized limitations of the language is the fact that the connection between the control flow and the data objects is under-specified [4, 19, 20]: items and their role in ruling the control flow are not expressed explicitly. This is one of the reasons why, as noticed also by other authors [2, 21], in some cases the approach is too general to capture the specificity of a particular domain or to perform automatic tasks. The language supports the representation of data objects and data stores (see Figure 2a) which however can only express that a certain information is needed for the activity to be executed (or as an output of the activity). To represent explicitly the items on which an activity is executed, the language should be enriched. For instance, in [21] the authors show that in order to reason about the data that is manipulated by a process, one necessarily needs to enrich the language with formal annotations about the way the activities manipulate the data. Another possibility supported by the languages is the representation of repetitive activities. BPMN allows for the representation of activities with a loop or multi-instance marker. However, also in this case, the items on which the loops or the activities need to be repeated is not represented explicitly [20].

In construction processes there is the need of representing locations and location-based relationships. Consider, for instance, the example in which the activity Lay Floor must be performed in each room of a floor before the activity Install Wooden Window can be performed in any room of the same floor. This is because in this way the company laying the floor can complete its work in one floor without obstructions by the other company. Figure 2a represents the case in which the locations are read from a data object or a data store. As can be seen, it is not possible to specify to which locations each activity applies and how the locations affect the control flow (in this case the precedence between the two tasks). Another possibility would be to represent a BPMN task as a pair of a construction activity and the location where to perform it, as represented in Figure 2b. This model describes that both activities need to be performed in floors 1 and 2 (f1, f1) in rooms 1, 2 and 3 (r1, r2 and r3). The precedences are defined in such a way that Install Wooden Window can start in a floor only after the execution of the task Lay Floor is completed in the three rooms of that floor. It is not hard to see that, already with few activities and locations the model becomes difficult to specify, to understand and thus also to maintain. It must also be considered that constraints can be more complicated than simple precedences. Consider, for instance, the case in which there is no order between laying the floor and the other activities, but once it is started in a floor, then no other tasks can be executed there. We refer to this as an *exclusive* constraint. To represent such a pattern in BPMN one would need to explicitly represent, among the possible executions, the allowed one, i.e., those in which the activity Lay Floor is not interrupted during its execution in a floor. Figure 3 reports this case considering one floor only



(a) BPMN Data Object (top) and Data Store (bottom)


Lay Floor <f1,r1>
Lay Floor <f1,r2>
Lay Floor <f1,r3>
Install Wooden Windows <f1,r1>
Install Wooden Windows <f1,r2>
Install Wooden Windows <f1,r3>
Lay Floor <f2,r1>
Lay Floor <f2,r2>
Lay Floor <f2,r3>
Install Wooden Windows <f2,r1>
Install Wooden Windows <f2,r2>
Install Wooden Windows <f2,r3>

(b) Precedence between Lay Floor and Install Wooden Window where locations are represented as part of the activity

Figure 2: Representing Activities and Locations in BPMN

and one additional activity beside Lay Floor. One can imagine that having more levels and more activities makes the model even more complicated. These complications are due to the characteristic of BPMN of being procedural. Procedural approaches define all and only the possible executions that are allowed, by specifying at each step what are the possible future steps. This makes them less flexible and adaptable, which is a requirement of construction processes to face unexpected events and changing requirements. For instance, let us consider again the situation depicted in Figure 3. If one wants to introduce the activity Install Aluminum Windows that can be performed at any time with respect to the installation of the wooden windows and the laying of the floor, but still respecting the exclusivity constraint at the floor for the Lay Floor activity, then the diagram in Figure 3 will have many more alternatives than it already has. Similarly, if new constraints must be added or existing constraints must be relaxed, then all the possible sequences must be analyzed and potentially changed [22].

Declarative approaches are discussed in the literature as approaches that better support flexibility [14, 23, 24]. Given that they represent only the constraints that an execution must satisfy, without representing all of them explicitly, when a new condition needs to be captured usually few new constraints need to be added. In this way the representation results to be more compact and the changes more circumscribed. It is however true that it might be more complicated to understand how to change the specification in order to obtain a desired result [25]. Declare [26] has been introduced as a declarative work flow management system that supports multiple declarative languages such as DecSerFlow [27] and ConDec [28]. DecSerFlow provides a number of constraint templates that are grounded on LTL and that have a graphical representation so as to easy a model specification. In Declare, expressing that Lay floor must be done before installing the wooden windows while installing the aluminum windows can be done at any time, can be done by just putting a precedence constraint between Lay Floor and Install Wooden Windows, and leave Install Aluminum Windows unconstrained. A precedence, indeed, does not represent a strict sequence, but a requirement on the ordering between two tasks meaning that other tasks can in principle be performed in-between the two.

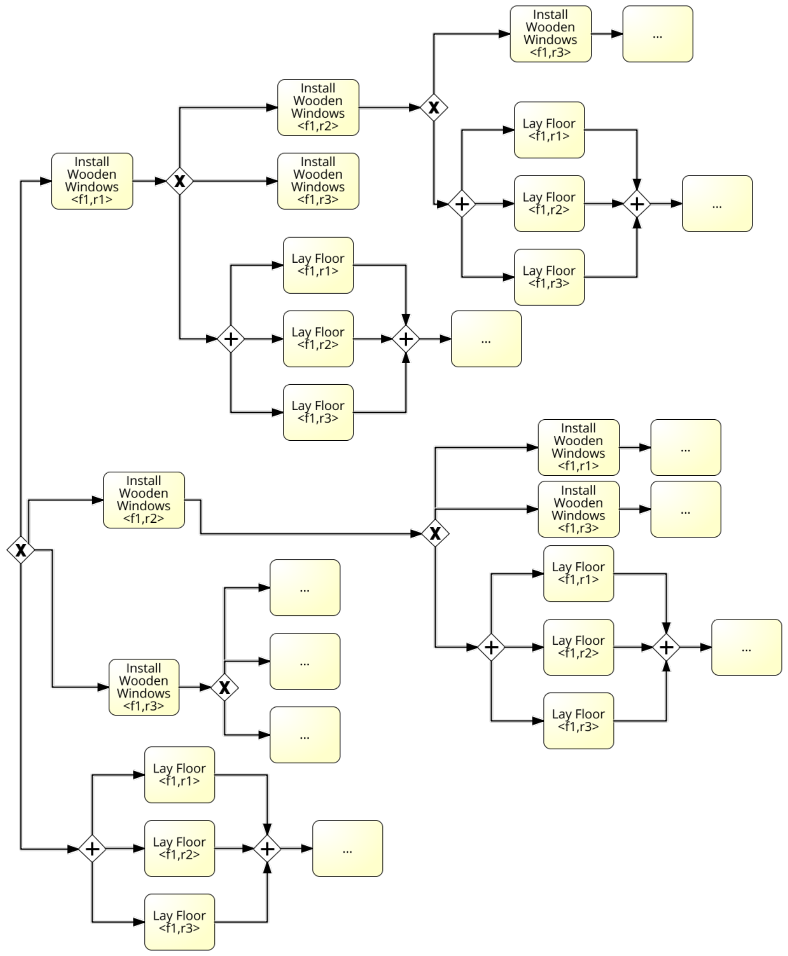

Install Wooden Windows <f1,r1>
Install Wooden Windows <f1,r2>
Install Wooden Windows <f1,r3>
Lay Floor <f1,r1>
Lay Floor <f1,r2>
Lay Floor <f1,r3>
...

Figure 3: Exclusivity constraint in BPMN for Lay Floor, without ordering constraints with Install Wooden Windows

However, similarly to BPMN, also Declare does not consider other perspectives besides the activities and the control flow [13]. Therefore, to represent the locations and location-based dependencies one would need to do something similar to Figure 2b and represent the locations as part of the activities. This would make a model more complicated than needed, difficult to understand and maintain. Additionally, in Declare it is not possible to express an exclusivity constrain. Let us consider again the example in Figure 3. An exclusivity constraint (without ordering constraints) is requiring that the installation of the window in one room, for each of the three rooms, either precedes the execution of Lay Floor (LF) in all of the three rooms, or succeeds it. Let us consider Install Wooden Window in room 1 at floor 1 ($\mathsf{IWW}\ \langle \mathsf{f1}, \mathsf{r1} \rangle$), then the constraint to express would be something like (and similarly for $\mathsf{IWW}\ \langle \mathsf{f1}, \mathsf{r1} \rangle$ and $\mathsf{IWW}\ \langle \mathsf{f1}, \mathsf{r3} \rangle$):

$$((\mathsf{LF}\ \langle \mathsf{f1}, \mathsf{r1} \rangle\ \textit{and}\ \mathsf{LF}\ \langle \mathsf{f1}, \mathsf{r2} \rangle\ \textit{and}\ \mathsf{LF}\ \langle \mathsf{f1}, \mathsf{r3} \rangle)\ \textit{precedes}\ \mathsf{IWW}\ \langle \mathsf{f1}, \mathsf{r1} \rangle) \quad \textit{or}$$
$$(\mathsf{IWW}\ \langle \mathsf{f1}, \mathsf{r1} \rangle\ \textit{precedes}\ (\mathsf{LF}\ \langle \mathsf{f1}, \mathsf{r1} \rangle\ \textit{and}\ \mathsf{LF}\ \langle \mathsf{f1}, \mathsf{r2} \rangle\ \textit{and}\ \mathsf{LF}\ \langle \mathsf{f1}, \mathsf{r3} \rangle)$$

In Declare it is not possible to express such alternatives on patterns. There are approaches in the literature that extend Declare in order to support more expressive conditions to rule the control flow. These will be discussed in the Related Work (Section 7), but basically they also fail in representing exclusive constraints.

3. CoPModL: Construction Process Modeling Language

One of the requirements identified in the previous section is the need of supporting *one-of-a-kind* projects. To do this we decouple two components of process modeling that we call *configuration* and *flow* part. The idea is that the former defines the parts of a process model that more likely can be reused from project to project, although extended and adapted as needed. This basically consists in the vocabulary of activities to be performed, the attributes defining the locations, the required crafts and so on. The flow part, instead, represents the dynamic part mainly capturing *i)* which activities, among those defined in the configuration part, are expected to be executed, *ii)* for each of them, in which locations to execute them and *iii)* the dependencies in performing the activities. In the following we will introduce the two parts in an intuitive way, providing some examples taken from or inspired by real projects, and propose a formalization.

3.1. Configuration Part

One of the core elements of process modeling are the activities, which represent pieces of work to be performed (e.g., install window, lay floor). Additionally, the items on which the activities will be executed need to be part of the model. On the one hand, the representation of items should be abstract, so that items can be easily specified and identified. On the other hand, the representation should accommodate different perspectives (e.g., skeleton and interior construction). To this aim, we foresee representations that consist of *i)* a hierarchy of *attributes*, and *ii)* for each attribute a range of possible values (called *domain values*). The attributes, their hierarchy, and the possible values will vary depending on the domain and on the project. For a construction project, one can represent in this way a location-break-down structure of a building.

Example. *A possible hierarchy to represent locations, as depicted in Figure 4, is composed of the following attributes:* i) sector *(*sr*), which represents an area of the construction site where activities can be performed more or less independently from the other sectors (like different buildings or wings); as possible values, in the running example that we will use in this paper we consider* B1 *and* B2 *(respectively standing for building 1 and 2);* ii) level *(*l*), identifying a floor; we consider 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). Note that in this work we are not concerned about where locations are physically located in the building, like, for instance, which location is adjacent to which other. This knowledge should come from some expert or from drawings of the building. BIM technologies can help in this respect, but this is out of the scope of this paper.*

The domain values of an attribute can be ordered. This is the case, for instance, for the levels where often there is the need of specifying an ascending ($\mathsf{u1} < \mathsf{f0} < \mathsf{f1}$, from bottom to top) or descending order.

We call *item structure* a hierarchy of attributes. As anticipated, in a construction project several perspectives (e.g., skeleton, interior) are involved for which different activities need to be performed on items having a different item structure. To express this, we define a set of perspectives and for each of them the corresponding *item structure*. An item structure is defined as a tuple of attributes, and a set of *item values* defining the allowed values for the items. Thus, an item is a sequence of values indexed by the attributes.

Example. *Common perspectives in construction are the* skeleton *and* interior *construction, requiring respectively a coarser representation of the locations and a more fine-grained one. The construction of the skeleton, indeed, occurs at the beginning of a construction process, and a coarse representation of the locations is usually sufficient since little parallelism between the trades occurs and the execution of the activities*

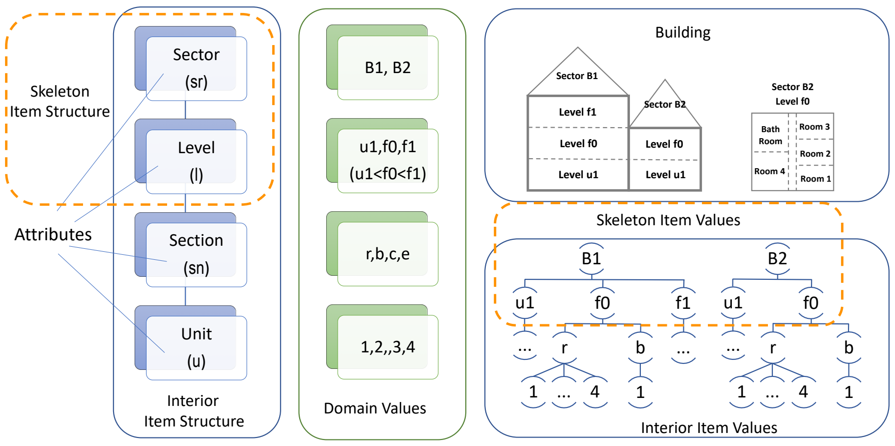


Figure 4: Representation of the items in the hotel case study.

takes long. One may foresee several suitable ways to identify the locations in this case. Among these, one possibility is to consider sectors and levels. Accordingly, in Figure 4, an item for the skeleton is described in terms of sector and level only, with sector B1 *having three levels and sector* B2 *only two. Accordingly, the item structure for the skeleton perspective is* $\langle \text{sector}, \text{level} \rangle$ *and the possible values for sector* B1 *are* $\langle B1, u1 \rangle$, $\langle B1, f0 \rangle$, $\langle B1, f1 \rangle$, *while for* B2 *they are* $\langle B2, u1 \rangle$, $\langle B2, f0 \rangle$.

The interior construction usually occurs simultaneously to other construction parts (such as the envelope), and concerns activities for the construction of the inner parts of the building. Here, locations should allow one to identify the technological content of an area, that is identified with the section. *Indeed, different sections require different activities to be performed there. For instance, the activities to be performed in a bathroom are different from the activities to be performed in a room, a kitchen or a swimming pool. As depicted in Figure 4, the item structure for the interior perspective is* $\langle \text{sector}, \text{level}, \text{section}, \text{unit} \rangle$ *and among the possible values for sector* B1 *there are* $\langle B1, f0, r, 1 \rangle, \ldots, \langle B1, f0, r, 4 \rangle$ *to identify the four rooms at level* f0 *of building* B1.

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 Perspectives, 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 perspective and $\mathcal{I}_v \subseteq \Sigma_{\alpha_1} \times ... \times \Sigma_{\alpha_n}$ is the set of *item values*.

3.2. Flow Part

The configuration part can be seen as the “vocabulary” containing possible activities and items. Based on them, the flow part describes the construction process starting from specifying on which items the activities must be executed. 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 perspective. We use $\langle \mathsf{a}, \mathtt{i} \rangle$ for an activity on one item $\mathtt{i}$.

A process model must also capture dependencies (ordering constraints) on the execution of the activities. Combining activity and locations in a task, allows us to refine the representation of the execution constraints, being more precise. In particular, one may want to order the locations according to some criteria (e.g., bottom to top, far from the entrance to close) and specify that an activity should be performed following that order.

Example. *Not surprisingly, in construction walls are built from bottom to top. Another common pattern is that cleaning is performed from top to bottom. Intuitively, this is because the cleaning of an upper floor* *makes the underlying floors dirty. One may also want to specify that the floor should be laid starting from the rooms that are farther from the entrance and proceeding towards it.*

A requirement from the domain is that when defining a dependency between two tasks it should be possible to clarify the *scope*, specifying whether it applies *i)* at the task level, that is, the first task must be finished in all its locations before progressing with the second; *ii)* at the item level, that is, once the first task is finished in a location the second can be performed in the same location; *iii)* or on some broader location to express, for instance, that only after the first task is finished everywhere in a floor, the second can start in the locations at the same floor. In the original language from which this approach started, the scope of precedences was provided as disambiguation notes in natural language.

Example. *Consider a precedence between the tasks* Lay Floor *and* Install Wooden Window*. A dependency at the level of task would require the floor to be finished everywhere before starting installing the windows. However, it is probably more likely to start with the installation of the windows once the floor is finished everywhere at a level. This can be achieved with a scope at the floor level. In some cases, it is even feasible to start with the installation in a room once the floor is finished there (i.e., at the scope of section).*

Besides refining the precedence dependencies between tasks specifying the scope, the construction domain requires the modeling of some different temporal ordering relations between tasks. Among these, from the previous project the need emerged to express that a task must have exclusive access to an area, that is no other tasks can be performed there while it is executed. By default in construction, two items cannot be performed at the same time on the same item (e.g., in the same room of a particular floor of a building). In some cases, however, there is the need to express an exclusive constraint at a higher level (for instance, no other task can be performed at the same floor where the exclusive task is executing). Additionally, there is the need to express exclusive constraints between two tasks, that is, the execution of the two cannot be interrupted by other tasks.

Example. *For safety reasons, once excavate starts in an area no other tasks can be performed there until it is finished. Moreover, once excavate is finished, before any other task can start, the area must be secured.*

Another type of constraint that is needed is to express a relation between two tasks such that the first task must be performed before the second, but in order for the first to progress on another item it also has to wait for the second one to be finished on the previous item.

Example. *Let us consider the construction of a building with several floors. In order to build the walls at a floor, the scaffolding must be installed at that floor first. However, the scaffolding for the next floor is anchored to the previous floor, that is why the installation of the scaffolding for a subsequent floor can be performed only after the concrete has been poured for the previous floor.*

In summary, the flow part defines a set of tasks that must be performed and dependencies to be satisfied. Formally, a flow part $\mathcal{F}$ is a tuple $\langle T, D\rangle$, where T and D are sets of tasks and dependencies.

In the original language, the dependencies were declarative, i.e., not expressing strict sequences but constraints to be satisfied. In this paper we extend the kind of dependencies that can be specified, we support the specification of a scope and we provide a formal semantics in terms of LTL over finite traces. A detailed description and the formalization are provided in the following.

The fact of being declarative supports the flexibility and adaptability of a process model [14, 23, 24], which is also supported by the distinction between *configuration* and a *flow* parts. Additionally, this distinction better supports the reusability (especially of the configuration) from project to project. With respects to the requirements identified in Section 2.1, the current approach is able to represent the main elements

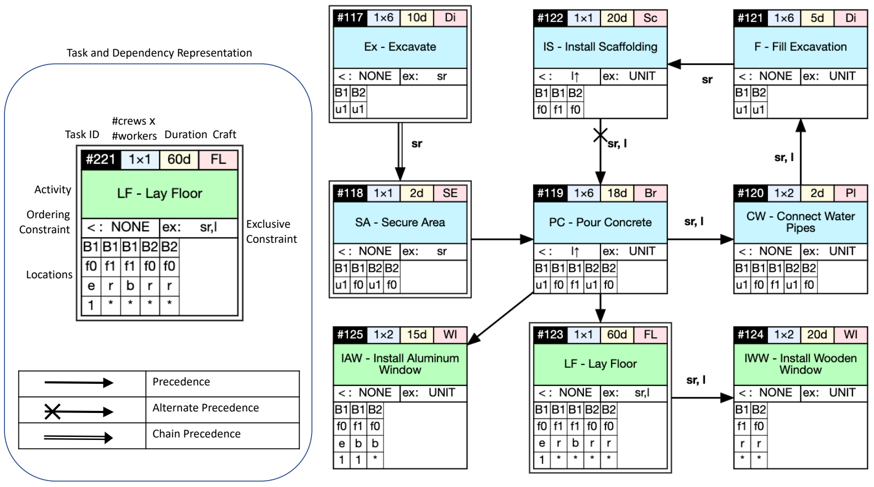


Figure 5: Excerpt of the hotel process model. (The * denotes all possible values for an attribute).

of a construction process model, that is different perspectives, activities, locations and location-based dependencies. Finally, its formal grounding makes the approach non-ambiguous and paves the way to the development of automatic tools, as we will show in Section 5.

3.3. Constraint Specification in CoPModL

CoPModL defines a number of constructs for the specification of execution and ordering constraints on task execution. In particular, we consider that the execution of an activity on an item does not occur more than once and that a task execution has a duration. Ordering constraints basically relate the start and the end of the tasks, for instance requiring the start of a task to occur after the end of another one. To this aim, for each task $\langle \mathsf{a}, i\rangle$ we introduce a start and an end event represented as $\mathsf{start}(\mathsf{a}, i)$ and $\mathsf{end}(\mathsf{a}, i)$ respectively. An execution is then a sequence of states, each of which comprising a set of start and end events for different tasks that may occur simultaneously. For each construct of the language we provide a collection of LTL formulas that capture the desired behavior by constraining the occurrence of the start and end events of the activities on the items.

To support the representation of a process model and provide an overall view of a process flow we propose a graphical representation. Figure 5 shows an excerpt of a process for the construction of a hotel. We will use it as a running example. Intuitively, each *box* is a task where the *color* represents the perspective, 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.

Example. *As an example, consider the task* Excavate *in Figure 5. The task belongs to the skeleton perspective (coded with a light blue color). Accordingly, the item structure is composed of two attributes: the sector and the level, which correspond respectively to the first and the second row of the matrix in the box. The item values are two in this case:* $\langle B1, u1\rangle$ *and* $\langle B2, u1\rangle$ *represented as columns and specifying that the task must be executed at the underground level (u1) of building 1 (B1) and building 2 (B2).*

Arrows between the boxes represent binary dependencies among the tasks and can be of different kinds. The dependency names and intuitive meaning are inspired by Declare [26], although the language and its

LTL$_f$ semantics are different. To improve readability, Table 1 lists macros that we use in the definition of the semantics of the CoPModL dependencies and Table 2 summarizes the dependencies, providing a short explanation for them and the formal semantics.

For the sake of clarity, we recall that the LTL formulas are built up starting from a set of propositions $\mathcal{P}$. In our case the propositions are of the form $\mathsf{start}(\mathsf{a}, \mathtt{i})$ or $\mathsf{end}(\mathsf{a}, \mathtt{i})$, where a ranges over all activities in a given model and $\mathtt{i}$ over all items for which a is foreseen. In addition to the standard operators $\wedge$, $\vee$, and $\neg$, LTL allows one to construct temporal expressions of the form $\Box\phi$, $\Diamond\phi$ and $\bigcirc\phi$, with the meaning that condition ϕ must be satisfied *i) always* in the future, *ii) eventually* in the future, and *iii)* in the *next* state, respectively. Furthermore, the formula $\phi \mathsf{U} \psi$ requires condition ϕ to be true until ψ becomes true [29].

The semantics is the one of LTL$_f$, that is, formulas are evaluated over finite traces as defined by De Giacomo and Vardi [11]. We restrict ourselves to finite traces because a construction project that runs forever would be absurd.

Execute. As described previously, the flow part specifies a set of tasks. Note that all tasks defined in the flow part need eventually to be executed. This is captured by specifying an execute dependency *executes*(t), for each task t of the flow part. Graphically, this is represented by drawing a box. Formally, it is expressed by an *executes*(t) constraint for $\mathsf{t} = \langle \mathsf{a}, I \rangle$. The constraint

$$\textit{executes}(\langle \mathsf{a}, I \rangle)$$

is a shorthand for a collection of LTL formulas that formalize four intuitive principles of how activities are to be executed on items. The four principles are the following:

Eventual start: For each item $\mathtt{i} \in I$, activity a has to start eventually:

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

Start-end order: Every start event is followed by an end event:

$$\forall \mathtt{i} \in I \quad \Box(\mathsf{start}(\mathsf{a}, \mathtt{i}) \rightarrow \bigcirc\Diamond\mathsf{end}(\mathsf{a}, \mathtt{i}).$$

Non-repetition: An activity cannot be executed more than once on the same item, that is, start and end events never repeat:

$$\forall \mathtt{i} \in I \quad \Box(\mathsf{start}(\mathsf{a}, \mathtt{i}) \rightarrow \bigcirc\Box\neg\mathsf{start}(\mathsf{a}, \mathtt{i})) \wedge \Box(\mathsf{end}(\mathsf{a}, \mathtt{i}) \rightarrow \bigcirc\Box\neg\mathsf{end}(\mathsf{a}, \mathtt{i})).$$

Non-concurrency: At a given point in time, no other task can be performed on an item $\mathtt{i}$ concurrently with a, i.e., for all tasks $\langle \mathsf{b}, I_\mathsf{b} \rangle$ distinct from $\langle \mathsf{a}, I \rangle$ in the set of tasks T of the model, the following holds:

$$\forall \mathtt{i} \in I \cap I_\mathsf{b} \quad \Box(\mathsf{start}(\mathsf{a}, \mathtt{i}) \rightarrow \neg\mathsf{start}(\mathsf{a}', \mathtt{i})\ \mathsf{U}\ \mathsf{end}(\mathsf{a}, \mathtt{i})).$$

These constraints are captured by the four macros *eventually_starts*$(\langle \mathsf{a}, I \rangle)$, *start-end_order*$(\langle \mathsf{a}, I \rangle)$, *non_repetition*$(\langle \mathsf{a}, I \rangle)$, and *non_concurrency*$(\langle \mathsf{a}, I_\mathsf{a} \rangle)$, respectively.

MACROS:

<u>*Eventual start*:</u> *eventually_starts*($\langle \mathsf{a}, I\rangle :$ task)
Activity a must be started on all instances in I
$\forall i \in I \quad \Diamond\, \mathsf{start}(\mathsf{a}, i)$
<u>*Start_end order*:</u> *start-end_order*($\langle \mathsf{a}, I\rangle :$ task)
Every start event is followed by an end event
$\forall i \in I \quad \Box(\mathsf{start}(\mathsf{a}, i) \rightarrow \bigcirc\Diamond \mathsf{end}(\mathsf{a}, i)$
<u>*Non-repetition*:</u> *non_repetition*($\langle \mathsf{a}, I\rangle :$ task)
An activity cannot be executed more than once on the same item
$\forall i \in I \quad \Box(\mathsf{start}(\mathsf{a}, i) \rightarrow \bigcirc\Box\neg \mathsf{start}(\mathsf{a}, i)) \wedge \Box(\mathsf{end}(\mathsf{a}, i) \rightarrow \bigcirc\Box\neg \mathsf{end}(\mathsf{a}, i))$
<u>*Non-concurrency*:</u> *non_concurrency*($\langle \mathsf{a}, I_\mathsf{a}\rangle :$ task)
No other task than a can be performed on an item of I_a at a given time
$\forall \langle \mathsf{b}, I_\mathsf{b}\rangle \in T \setminus \{\langle \mathsf{a}, I_\mathsf{a}\rangle\}$
$\forall i \in I_\mathsf{a} \cap I_\mathsf{b} \quad \Box(\mathsf{start}(\mathsf{a}, i) \rightarrow \neg \mathsf{start}(\mathsf{b}, i)\ \mathsf{U}\ \mathsf{end}(\mathsf{a}, i))$
<u>*Precedes*:</u> *precedes*($\langle \mathsf{a}, I_\mathsf{a}\rangle :$ task, $\langle \mathsf{b}, I_\mathsf{b}\rangle :$ task)
Activity a must be performed on I_a before activity b is performed on I_b
$\forall i_\mathsf{a} \in I_\mathsf{a}, i_\mathsf{b} \in I_\mathsf{b} \quad \neg \mathsf{start}(\mathsf{b}, i_\mathsf{b})\ \mathsf{U}\ \mathsf{end}(\mathsf{a}, i_\mathsf{a})$
<u>*Not interrupt*:</u> *not_interrupt*($T_1 :$ set of tasks, $T_2 :$ set of tasks)
All tasks in T_1 must not be interrupted by the tasks in T_2 and vice versa, i.e., all tasks in T_1 are performed on all their instances before any task in T_2, or the other way around.
$\forall \mathsf{t}_1 \in T_1, \mathsf{t}_2 \in T_2 \quad$ *precedes*$(\mathsf{t}_1, \mathsf{t}_2)$ or $\forall \mathsf{t}_1 \in T_1, \mathsf{t}_2 \in T_2 \quad$ *precedes*$(\mathsf{t}_2, \mathsf{t}_1)$

Table 1: CoPModL Macros for Process Modeling.

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.

Example. *Consider the task Pour Concrete (PC) in Figure 5. To specify that it must be performed from bottom to top, we graphically use the label* $<:\mathsf{l}\uparrow$*, which corresponds to the constraint*

$$\textit{ordered\_execution}(\langle \mathsf{PC}, I\rangle, \mathsf{l}\uparrow),$$

meaning that the items I are partitioned according to their values for 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, the following formula captures that b cannot start on an item i_b until a is not finished on an instance i_a, for all items in I_a and I_b:

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

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.

EXECUTION CONSTRAINT:

EXECUTE: *executes*($\langle \mathsf{a}, I\rangle$: task)

Activity a must be started on all instances in I, every start event must be followed by an end event, activity a cannot be repeated on any item, and at any time at most one task can be performed on an item

$$\textit{eventually\_starts}(\langle \mathsf{a}, I\rangle) \quad \text{and} \quad \textit{start-end\_order}(\langle \mathsf{a}, I\rangle) \quad \text{and}$$
$$\textit{non\_repetition}(\langle \mathsf{a}, I\rangle) \quad \text{and} \quad \textit{non\_concurrency}(\langle \mathsf{a}, I\rangle)$$

UNARY AND BINARY DEPENDENCIES:

ORDERED EXECUTION: *ordered_execution*($\langle \mathsf{a}, I\rangle$: task, $\mathcal{O}$: ordering)

Activity a must be executed on all instances in I following the order in $\mathcal{O}$

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

EXCLUSIVE EXECUTION: *exclusive_execution*($\langle \mathsf{a}, I_\mathsf{a}\rangle$: task, s : scope)

The execution of activity a cannot be interrupted by other activities on instances at the same scope s

$$\textit{executes}(\langle \mathsf{a}, I_\mathsf{a}\rangle) \quad \text{and}$$
$$\forall \langle \mathsf{b}, I_\mathsf{b}\rangle \in T \setminus \{\langle \mathsf{a}, I_\mathsf{a}\rangle\}, \ \forall \pi_\mathsf{s} \in \Pi_\mathsf{s}(I_\mathsf{a}), \ \forall i_\mathsf{b} \in \sigma_{\pi_\mathsf{s}}(I_\mathsf{b})$$
$$\textit{not\_interrupt}(\{\langle \mathsf{a}, \sigma_{\pi_\mathsf{s}}(I_\mathsf{a})\}\rangle, \{\langle \mathsf{b}, \{i_\mathsf{b}\}\rangle\})$$

PRECEDENCE: *precedence*($\langle \mathsf{a}, I_\mathsf{a}\rangle$: task, $\langle \mathsf{b}, I_\mathsf{b}\rangle$: task, s : scope)

Activity a must be executed before activity b on instances at the same scope s

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

ALTERNATE PRECEDENCE: *alternate*($\langle \mathsf{a}, I_\mathsf{a}\rangle$: task, $\langle \mathsf{b}, I_\mathsf{b}\rangle$: task, s : scope)

Activity a precedes activity b and once b is started on an instance, a cannot progress on an instance at a different scope until b is finished

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

CHAIN PRECEDENCE: *chain*($\mathsf{t}_1 = \langle A, I_A\rangle$: task, $\mathsf{t}_2 = \langle B, I_B\rangle$: task, s : scope)

Activity a precedes activity b and their executions cannot be interrupted by other activities on the instances at the same scope s

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

Table 2: CoPModL Unary and Binary Dependencies for Process Modeling.

Exclusive execution. An *exclusive execution* constraint *exclusive_execution*$(\langle \mathsf{a}, I_{\mathsf{a}}\rangle, \mathsf{s})$ expresses that once an activity is started 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:

$$\begin{aligned} &\textit{executes}(\langle \mathsf{a}, I_{\mathsf{a}}\rangle) \quad \text{and} \quad \forall \langle \mathsf{b}, I_{\mathsf{b}}\rangle \in T \setminus \{\langle \mathsf{a}, I_{\mathsf{a}}\rangle\},\ \forall \pi_{\mathsf{s}} \in \Pi_{\mathsf{s}}(I_{\mathsf{a}}), \forall \mathtt{i}_{\mathsf{b}} \in \sigma_{\pi_{\mathsf{s}}}(I_{\mathsf{b}}) \\ &\qquad\qquad\qquad\qquad\quad \textit{not\_interrupt}(\{\langle \mathsf{a}, \sigma_{\pi_{\mathsf{s}}}(I_{\mathsf{a}})\rangle\}, \{\langle \mathsf{b}, \{\mathtt{i}_{\mathsf{b}}\}\rangle\}) \end{aligned}$$

In this formula, the result of $\Pi_{\mathsf{s}}(I_{\mathsf{a}})$ 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*. 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 5, 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.

Example. *In Figure 5, the dependency between* pour concrete *and* lay floor *is at task level, while the one between* lay floor *and* install wooden window *is labeled with "sr,l", to represent the scope* $\langle \textit{sector}, \textit{level}\rangle$*: given a sector and a level, the activity* lay floor *must be done in every section and unit before* install wooden window *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.

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

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 Install Scaffolding and the Pour Concrete: 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 $(\sigma_{\pi'_{\mathsf{s}}}(I_{\mathsf{a}}))$:

$$\begin{gathered}
\mathit{precedence}(\langle \mathsf{a}, I_{\mathsf{a}}\rangle, \langle \mathsf{b}, I_{\mathsf{b}}\rangle, \mathsf{s}) \quad \text{and} \quad \forall \pi_{\mathsf{s}}, \pi'_{\mathsf{s}} \in \Pi_{\mathsf{s}}(I_{\mathsf{a}}) \cap \Pi_{\mathsf{s}}(I_{\mathsf{b}}), \pi_{\mathsf{s}} \neq \pi'_{\mathsf{s}} \\
\mathit{not\_interrupt}(\{\langle \mathsf{a}, \sigma_{\pi_{\mathsf{s}}}(I_{\mathsf{a}})\rangle, \langle \mathsf{b}, \sigma_{\pi_{\mathsf{s}}}(I_{\mathsf{b}})\rangle\}, \{\langle \mathsf{a}, \sigma_{\pi'_{\mathsf{s}}}(I_{\mathsf{a}})\rangle, \langle \mathsf{b}, \sigma_{\pi'_{\mathsf{s}}}(I_{\mathsf{b}})\rangle\})
\end{gathered}$$

Note that 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.

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 excavate 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 $\mathit{chain}(\langle \mathsf{a}, I_{\mathsf{a}}\rangle, \langle \mathsf{b}, I_{\mathsf{b}}\rangle, \mathsf{s})$, 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,

$$\begin{gathered}
\mathit{precedence}(\langle \mathsf{a}, I_{\mathsf{a}}\rangle, \langle \mathsf{b}, I_{\mathsf{b}}\rangle, \mathsf{s}) \quad \text{and} \quad \forall \pi_{\mathsf{s}} \in \Pi_{\mathsf{s}}(I_{\mathsf{a}}) \cap \Pi_{\mathsf{s}}(I_{\mathsf{b}}), \forall\, \mathsf{t}_3 = \langle \mathsf{c}, I_{\mathsf{c}}\rangle \in T \setminus \{\mathsf{t}_1, \mathsf{t}_2\} \\
\forall \mathtt{i} \in \sigma_{\pi_{\mathsf{s}}}(I_{\mathsf{c}}) \;\; \mathit{not\_interrupt}(\{\langle \mathsf{c}, \{\mathtt{i}\}\rangle\}, \{\langle \mathsf{a}, \sigma_{\pi_{\mathsf{s}}}(I_{\mathsf{a}})\rangle, \langle \mathsf{b}, \sigma_{\pi_{\mathsf{s}}}(I_{\mathsf{b}})\rangle\})
\end{gathered}$$

Graphically, it is represented as a double border arrow.

In the next section we present an excerpt of a real project, modeled according to CoPModL.

4. Process Modeling for the Hotel Scenario

The original model of the hotel case study consists of roughly fifty tasks. From that we extracted an excerpt and elaborated it by removing the ambiguities and modeling the requirements expressed as annotations in natural language using the construct of CoPModL. The excerpt is reported in Figure 5. It shows activities of the skeleton (blue) and interior perspectives (green). The item structure for the skeleton perspective 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 4).

According to the model, the task Excavate belongs to the skeleton perspective and must be performed in 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 floors 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* at scope sr (sector), graphically represented as a double bordered arrow annotated with the label sr. Additionally, the excavation cannot be interrupted by other activities while it is performed in a sector. This is expressed with an *exclusive execution* at scope sector, represented as double border box and a label *"ex:* sr*"*. The same condition is expressed for the task Secure Area.

Only after the area has been secured everywhere, the Pour Concrete task 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* represented with the label "$<: \mathsf{l} \uparrow$". Note that this task has an exclusive constraint *"ex:* UNIT*"*, i.e. an exclusive constraint at the finest granularity level for the locations, which expresses that two activities cannot be performed simultaneously in the same location. This is because by default it is assumed that two activities cannot be performed at the same time on the same item. This is then captured with an exclusive constraint which, being the default, is not graphically highlighted with double borders. An exclusive constraint is graphically highlighted only 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 (Connect Water Pipes) before the excavation is filled (Fill Excavation), and after this the task Install Scaffolding can start, proceeding from the bottom floor to the top. Once the scaffolding is installed at one level, not only the concrete *can* be poured at that level but it *must* be poured there before the scaffolding can be installed at the next level. In other words, Install Scaffolding needs to be performed before Pour Concrete but its execution is also conditioned by this latter. This requirement is captured by an *alternate precedence* at scope "sr, l". It is graphically represented with an arrow starting with an X.

When the pour concrete task is finished everywhere, the Lay Floor task can start (which belongs to the interior perspective). This task must be performed before Install Wooden Window, which is foreseen in all rooms, so not to damage them. This is captured by a precedence constraint between Lay Floor and Install Wooden Window 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 [30, 31] which we used to produce Figure 5 and is described in Section 6.

5. Satisfiability Checking

straightforward developing a tool requiring inputs from the user, one cannot assume that the provided input will be free of contradictions. Due to the logic semantics of CoPModL, we can check for the absence of contradictions by checking whether a given model is *satisfiable*, that is, there exists at least one trace satisfying it.

In this section, we will show first that for CoPModL constraints there is no difference between satisfiability over finite traces and over infinite traces, that is, a CoPModL model is satisfiable under LTL$_f$ semantics if and only if it is satisfiable under LTL semantics. This allows us in principle to perform satisfiability tests using model checking techniques. As an alternative, we develop a specially tailored graph-based algorithm. With experiments, we show that the new algorithm outperforms the model checker by several orders of magnitude in checking the satisfiability of CoPModL models.

5.1. Equivalence of LTL and LTL$_f$ Satisfiability for CoPModL

To formally specify the trace semantics of LTL and LTL$_f$ we draw upon the notation of De Giacomo and Vardi, who first studied the formal properties of LTL$_f$ [11].

Let $\mathcal{P}$ be a fixed set of propositions. An infinite trace is a mapping $\tau\colon \mathbb{N} \to 2^{\mathcal{P}}$, and a finite trace is a mapping $\tau\colon \mathbb{N}_n \to 2^{\mathcal{P}}$, where $\mathbb{N}_n = \{0, \dots, n-1\}$ is the set of the first n natural numbers for some natural number n. In this case we say that τ is of length n. Intuitively, $\mathbb{N}$ or $\mathbb{N}_n$, respectively are the sets of states of the trace τ, and τ maps each state, represented by a natural number j, to the set of propositions $\tau(j)$ that hold in that state. If τ is a trace, τ is defined for j, and p is a proposition, then $\tau, j \models p$ if and only if $p \in \tau(j)$. We then say that p is satisfied in state j of τ. For the satisfaction of an arbitrary formula ϕ in a state of a trace we refer to [11]. We say that τ *satisfies* ϕ if $\tau, 0 \models \phi$. We say that ϕ is *satisfiable* over finite or infinite traces if there is a finite or infinite trace, respectively, that satisfies ϕ.

We recall that a *process model* is a pair $\mathcal{M} = (\mathcal{C}, \mathcal{F})$, where $\mathcal{C}$ is a configuration part and $\mathcal{F}$ is a flow part. The constraints of our language CoPModL are shorthands for LTL formulas whose propositions are of the form $\mathsf{start}(\mathsf{a}, \mathtt{i})$ or $\mathsf{end}(\mathsf{a}, \mathtt{i})$ for some activity a and item $\mathtt{i}$. Next we show that the constraints are satisfied only by traces of a specific, simple form. Moreover, satisfying traces share their characteristics independently of whether they are finite or infinite. By a slight abuse of language, we identify a process model $\mathcal{M}$ in the following with the constraints in $\mathcal{M}$.

Proposition 2. *Let $\mathcal{M}$ be a process model and τ be a finite or infinite trace that satisfies $\mathcal{M}$. Then the following holds:*

1. *For every proposition p there is exactly one state j such that $p \in \tau(j)$.*
2. *If $\tau, j \models \mathsf{start}(\mathsf{a}, \mathtt{i})$ and $\tau, k \models \mathsf{end}(\mathsf{a}, \mathtt{i})$, then $j < k$.*
3. *There is a least number $s \in \mathbb{N}$ such that if $\tau, j \models p$ for some proposition p and state $j \in \mathbb{N}$, then $j < s$.*

Proof. 1. Due to the Eventual Start principle, every start atom is satisfied in some state of τ and due to the Start-end Order principle, also the corresponding end atom is satisfied by some state. The Non-repetition principle ensures that there is only one satisfying state for each atom.

2. The Start-end Order principle for each activity on an item enforces that the state satisfying the start atom comes strictly before the one satisfying the corresponding end atom.

3. The third claim is an immediate consequence of the first claim, because there are only finitely many propositions. The number s is therefore the greatest index j for which $\tau(j-1) \neq \emptyset$ holds. □

We call the number s with the property of Claim 3 of Proposition 2 the *support* of τ. If τ is an infinite trace, then we define $\tau_{|s}$ as the finite trace that is obtained by restricting τ to the set $\mathbb{N}_s$. If τ is a finite trace of length n, then we define τ_∞ as the infinite trace that is obtained by extending τ to $\mathbb{N}$ with $\tau_\infty(j) = \emptyset$ for all $j \geq n$.

Proposition 3. *Let $\mathcal{M}$ be a process model and τ be a trace such that $\tau \models \mathcal{M}$.*

1. *If τ is finite, then also $\tau_\infty \models \mathcal{M}$;*
2. *If τ is infinite, then also $\tau_{|s} \models \mathcal{M}$, where s is the support of τ.*

Proof. Let $\mathcal{P}_\mathcal{M}$ be the set of atoms of $\mathcal{M}$. Among the constraints in $\mathcal{M}$ we distinguish between *execution constraints* on the one hand, which are those comprised by the *executes* constraints, and the formulas corresponding to the dependencies on the other hand. We denote the first set of formulas as $E_\mathcal{M}$ and the second as Δ_M.

Suppose that τ is finite and satisfies $E_\mathcal{M}$. Then clearly also τ_∞ satisfies $E_\mathcal{M}$, which is easy to verify by checking each type of existence formula. Conversely, suppose that τ is infinite and let s be the support of τ. Then it is again straightforward to verify that $\tau_{|s}$ satisfies $E_\mathcal{M}$, checking the four cases of existence formulas.

Regarding the formulas in Δ_M, we note that all of them are constructed from *elementary formulas*, which have the form $\delta = \neg\mathsf{start}(\mathsf{b}, \mathtt{i}_\mathsf{b}) \;\mathsf{U}\; \mathsf{end}(\mathsf{a}, \mathtt{i}_\mathsf{a})$. More precisely, all formulas in $\Delta_\mathcal{M}$ are obtained by taking conjunctions and disjunctions of elementary formulas. Clearly, "Precedes" dependencies are conjunctions of elementary formulas, while "Precedence " and "Ordered Execution" constraints are conjunctions of "Precedes" dependencies. A "Not Interrupt" dependency is a disjunction of conjunctions of "Precedes" dependencies. Finally, "Exclusive Execution", "Alternate Precedence", and "Chain Precedence" are conjunctions of "Not Interrupt" dependencies, possibly conjoined with "Precedence" dependencies.

We now show our claim about the constraints in $\Delta_{\mathcal{M}}$ by proving it for formulas that are repeated conjunctions and disjunctions of elementary formulas.

Suppose that τ is finite and satisfies the elementary formula δ. That means, $\mathsf{start}(\mathsf{b}, \mathtt{i}_{\mathsf{b}})$ is false until $\mathsf{end}(\mathsf{a}, \mathtt{i}_{\mathsf{a}})$ is true. Since there is only one state in τ where $\mathsf{end}(\mathsf{a}, \mathtt{i}_{\mathsf{a}})$ is true, it is immediate to check that also τ_∞ satisfies δ. If τ is infinite and satisfies δ, then one shows with the same argument that also $\tau_{|s}$ satisfies δ.

For arbitrary formulas, obtained by taking conjunctions and disjunctions, we show the claim by induction. Therefore, let again τ be finite and $\phi = \phi_1 \wedge \phi_2$. Suppose that $\tau \models \phi$. Then $\tau \models \phi_1$ and $\tau \models \phi_2$. By the induction hypothesis, also $\tau_\infty \models \phi_1$ and $\tau_\infty \models \phi_2$. Hence $\tau_\infty \models \phi$. The argument for the opposite direction and for disjunctions is similar. This completes the proof. □

Corollary 4. *Let $\mathcal{M}$ be a process model. Then $\mathcal{M}$ is satisfiable over finite traces if and only if $\mathcal{M}$ is satisfiable over infinite traces.*

Proof. Suppose that $\tau \models \mathcal{M}$. By the preceding proposition, if τ is finite, then also $\tau_\infty \models \mathcal{M}$, and if τ is infinite, then also $\tau_{|s} \models \mathcal{M}$, where s is the support of τ. □

Due to this equivalence, we can check satisfiability of a process model by submitting the corresponding formulas to an LTL model checker. However, this will usually come with a considerable overhead. For instance, in our experiment (Section 5.4), it took a state-of-the-art model checker more than 2 minutes to check a satisfiable model with 8 tasks and 9 dependencies.

5.2. Graph-based Satisfiability Checking

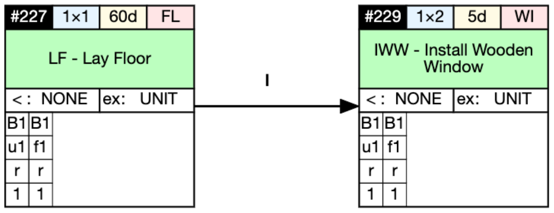
As an alternative, we will develop a satisfiability checking algorithm for CoPModL models that exploits the restricted form of such formulas. First, we discuss that it is enough to look only for satisfying traces of a special kind. Then we show that to find them we can inspect graphs derived from the model constraints.

In the following we call a finite trace that satisfies the *executes* constraints of $\mathcal{M}$ an *execution* of $\mathcal{M}$. An execution τ of $\mathcal{M}$ is *instantaneous* if for all activities a and items $\mathtt{i}$ it is the case that $\tau, j \models \mathsf{start}(\mathsf{a}, \mathtt{i})$ implies $\tau, j+1 \models \mathsf{end}(\mathsf{a}, \mathtt{i})$, that is, an activity on an item finishes immediately after it has started. An execution is *sequential* if no activities take place at the same time, that is, an activity is only started if all previously started activities have ended and no two activities start at the same time. We show next that when checking for satisfiability, it is enough to concentrate on sequential instantaneous executions.

Proposition 5. *Let $\mathcal{M}$ be a satisfiable process model. Then there exists a sequential instantaneous execution of $\mathcal{M}$.*

Proof. Let $\tau \models \mathcal{M}$ be an execution of $\mathcal{M}$. We transform τ into an instantaneous execution τ_{in} by anticipating the end of each activity to the state immediately after its start. That is, if $\tau, j \models \mathsf{start}(\mathsf{a}, \mathtt{i})$ and $\tau, k \models \mathsf{end}(\mathsf{a}, \mathtt{i})$, then we add $\mathsf{end}(\mathsf{a}, \mathtt{i})$ to $\tau_{in}(j+1)$ and remove it from $\tau_{in}(k)$. It is straightforward to check that all constraints continue to hold. Intuitively, this is the case because all constraints either relate to the unique existence of start and end events, or are essentially end-start relationships, which continue to hold if the end is anticipated.

Next, suppose that $\tau \models \mathcal{M}$ is an instantaneous execution of $\mathcal{M}$. We transform τ into a sequential execution τ_{seq} by first ordering activity-item pairs according to their start times, and then arbitrarily ordering pairs with the same start time. The execution τ_{seq} then alternates between start and end events for the pairs according to their ordering. Again, after this change all constraints continue to hold because, intuitively, the existence constraints and all end-start relationships are preserved. □



(a) A process model diagram $\mathcal{M}$.

$$\begin{tikzcd} \langle \mathsf{LF}, \langle \mathsf{B1}, \mathsf{u1}, \mathsf{r}, 1\rangle\rangle \arrow[d] & \langle \mathsf{LF}, \langle \mathsf{B1}, \mathsf{f1}, \mathsf{r}, 1\rangle\rangle \arrow[d] \\ \langle \mathsf{IWW}, \langle \mathsf{B1}, \mathsf{u1}, \mathsf{r}, 1\rangle\rangle & \langle \mathsf{IWW}, \langle \mathsf{B1}, \mathsf{f1}, \mathsf{r}, 1\rangle\rangle \end{tikzcd}$$

(b) The corresponding AI-graph $\mathcal{G}_{\mathcal{M}}$.

Figure 6: Example of (a) a process model diagram and (b) the corresponding AI-graph representation.

The algorithm we develop will rely 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 $\langle \mathsf{a}, \mathtt{i}\rangle$, conceptually representing the execution of a on $\mathtt{i}$. Ordering constraints are then represented as arcs in the graph. This allows us to characterize the satisfiability of a model by the absence of cycles in the corresponding AI graph.

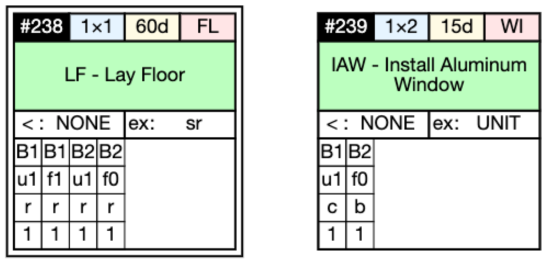
Satisfiability with precedences and ordered execution.. Let us first consider *P-models*, which are process models with precedence and ordered execution dependencies only— in addition to the execution constraints, which are present for every task. Given a P-model $\mathcal{M}$, we denote the corresponding AI-graph as $\mathcal{G}_{\mathcal{M}} = \langle V, A\rangle$, where for each task $\langle \mathsf{a}, I\rangle$ in the flow part and for each item $\mathtt{i} \in I$ there is an AI node $\langle \mathsf{a}, \mathtt{i}\rangle \in V$, without duplicates; for each precedence and ordered execution we introduce a number of arcs in A among the AI nodes in V. For instance, a precedence constraint between two tasks at the task scope is translated to 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.

Example. *Figure 6a represents a model with two tasks, where each activity needs to be performed in the locations $\langle B1, u1, r, 1\rangle$ and $\langle B1, f1, r, 1\rangle$. The precedence constraint is at scope level, requiring that* Lay Floor *is performed before* Install Wooden Window *at each floor. This translates to an AI graph with four nodes (as represented in Figure 6b) and an arrow between the AI nodes having the same level (thus between $\langle LF, \langle B1, u1, r, 1\rangle\rangle$ and $\langle IWW, \langle B1, u1, r, 1\rangle\rangle$ and between $\langle LF, \langle B1, f1, r, 1\rangle\rangle$ and $\langle IWW, \langle B1, f1, r, 1\rangle\rangle$).*

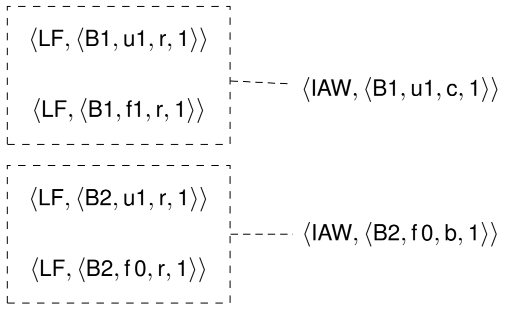
Theorem 6. *A P-model $\mathcal{M}$ is satisfiable if and only if the graph $\mathcal{G}_{\mathcal{M}}$ is cycle-free.*

Proof. If $\mathcal{G}_{\mathcal{M}}$ does not contain cycles, the nodes can be topologically ordered and the order can be translated into a sequential instantaneous execution that satisfies 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. □

Satisfiability for all models. 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 on items at the same scope. Considering an activity b to be executed on an item $\mathtt{i}_{\mathsf{b}}$ at the same scope (i.e., $\Pi_{\mathsf{s}}(\mathtt{i}_{\mathsf{b}}) = \pi_{\mathsf{s}}$), 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.



(a) A process model diagram $\mathcal{M}$.



(b) The corresponding DAI-graph $\mathcal{D}_{\mathcal{M}}$.

Figure 7: Example of (a) a process model with an exclusive constraint and (b) the corresponding DAI-graph.

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).

Example. *In the example in Figure 7a the task Lay Floor has an exclusive constraint at scope sector, meaning that once it is started in a sector it cannot be interrupted by other tasks. This is represented in the graph in Figure 7b by grouping in* exclusive groups *the AI nodes in the same sector (B1 and B2).*

The task Install Aluminum Window is also foreseen in the same sectors of Lay Floor, thus can potentially interrupt its exclusive execution. Therefore, in the graph the AI node $\langle$IAW, $\langle$B1, u1, c, 1$\rangle\rangle$ is linked with an undirected edge to the exclusive group for the activity lay floor in sector B1 (similarly for B2). Finding an orientation for the undirected edge corresponds to substituting it with one directed arc for each node in the group such that each arc has the same orientation, that is towards $\langle$IAW, $\langle$B1, u1, c, 1$\rangle\rangle$ or coming from it.

More formally, to represent a *not_interrupt* constraint in a graph we introduce *disjunctive activity-item* graphs (DAI-graphs) inspired by Fortemps and Hapke [32]. Given a *G-model* $\mathcal{M}$, the corresponding DAI-graph is $\mathcal{D}_{\mathcal{M}} = \langle V, A, X, E\rangle$, where $\langle V, A\rangle$ is defined as in the AI-graph of a P-model,[1] $X \subseteq 2^V$ is the set of exclusive groups, and $E \subseteq X \times V$ is a set of undirected edges, called *disjunctive edges*, connecting exclusive groups to single nodes. 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 7. *A G-model $\mathcal{M}$ is satisfiable iff the DAI-graph $\mathcal{D}_{\mathcal{M}}$ is orientable.*

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

[1]Including also the directed arcs to represent the precedence constraints of the chain and alternate dependencies (see the formalization in Section 3.3)

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

5.3. An Algorithm for Satisfiability Checking

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 four observations.

1. Cycles. If the graph $\mathcal{G}_\mathcal{M}$ obtained from $\mathcal{D}_\mathcal{M}$ by ignoring the undirected edges has a cycle, then also $\mathcal{D}_\mathcal{M}$ is not orientable.

2. 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.

Lemma 8. *Let $\mathcal{D}_\mathcal{M} = \langle V, A, X, E\rangle$ be a DAI-graph, and let $\mathcal{D}'_\mathcal{M} = \langle V, A, X, E \setminus E_1\rangle$ be the result of removing all simple edges E_1 from E. Then $\mathcal{D}_\mathcal{M}$ is orientable iff $\mathcal{D}'_\mathcal{M}$ is orientable.*

Proof. If $\mathcal{D}_\mathcal{M}$ is orientable it has an acyclic orientation. By topologically ordering it, we obtain an instantaneous sequential execution satisfying all the constraints. Since $\mathcal{D}'_\mathcal{M}$ is a subset of $\mathcal{D}_\mathcal{M}$, the topological order corresponds also to a satisfying execution of $\mathcal{D}'_\mathcal{M}$.

Conversely, if $\mathcal{D}'_\mathcal{M}$ is orientable, then there is a sequential execution that satisfies all constraints in $\mathcal{D}'_\mathcal{M}$. Since this execution imposes an ordering on all AI nodes, we can orient all simple edges according to this ordering. Hence, there is a possible orientation for the simple edges that does not introduce cycles. Such a sequential execution satisfies all constraints in $\mathcal{D}_\mathcal{M}$, and hence $\mathcal{D}_\mathcal{M}$ is orientable. □

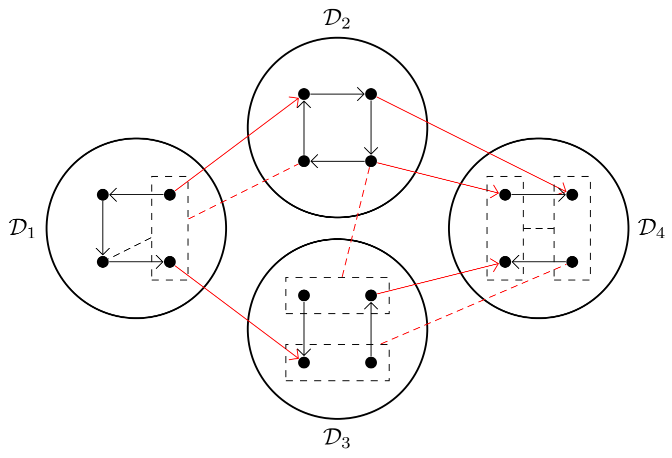
3. Resolving. Consider an undirected edge between a node u and an exclusive group of nodes $G \in X$. If there is a directed path from u to a node $v \in G$ (or the other way round), then there is only one way to orient the undirected edge between u and G without introducing cycles. For such an edge from u to G, we say that we *resolve* the edge if we *i)* check for directed paths between u and nodes in G, and *ii)* orient the edge in the direction of the path.

4. 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-subgraphs is orientable. Then each such subgraph can be checked independently.

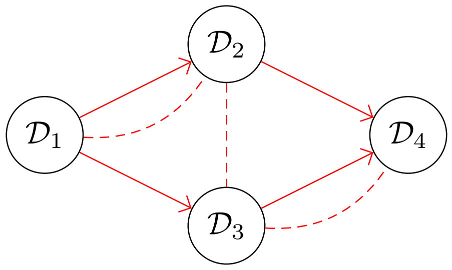
Let us discuss such partitioning in more detail. Given $\mathcal{D}_\mathcal{M} = \langle V, A, X, E\rangle$, let $P = \{S_1, \ldots, S_n\}$ be a partition of the node set V into disjoint subsets. Such a partition induces a canonical *quotient graph* $\mathcal{D}'_\mathcal{M}$ of $\mathcal{D}_\mathcal{M}$ as follows: *i)* the nodes of the quotient graph $\mathcal{D}'_\mathcal{M}$ are the partition sets; *ii)* there is an arc from S_i to S_j in $\mathcal{D}'_\mathcal{M}$ if there are nodes $u_i \in S_i$ and $u_j \in S_j$ such that there is arc from u_i to u_j in $\mathcal{D}_\mathcal{M}$; *iii)* an exclusive group G' of the quotient is obtained from an exclusive group $G \in X$ by collecting all sets S_i such that $S_i \cap G \neq \emptyset$ (then we say that G' is *induced* by G); *iv)* for every edge from an exclusive group G to a point u in E, the quotient has an edge from the induced group G' to the partition set S_i that contains u.

We say that such a partition is *acyclic* if the corresponding quotient graph has no cycles among its arcs. The quotient may still have edges that need to be oriented. However, we know from Lemma 8 that the task to orientate edges becomes trivial for simple edges.

Therefore, we are particularly interested in partitions where all edges are simple. Clearly, this is the case if and only if all exclusive groups in the quotient are singletons, which holds exactly if for every exclusive group $G \in X$, there is a partition set S_i such that $G \subseteq S_i$. We say that such a partition is a *proper partition.*



(a) DAI-graph partition.



(b) Subgraphs collapsed to single nodes.

Figure 8: Example of (a) a proper acyclic partition of a DAI-graph $\mathcal{D}_\mathcal{M}$ partitioned into subgraphs $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, $\mathcal{D}_4$. The partition is acyclic because, collapsing each subgraph into a single node, as shown in (b), yields an acyclic DAI-graph.

As an example, the partition in Figure 8a is an acyclic proper partition: *i)* there are no nodes belonging to the same exclusive group of $\mathcal{D}_\mathcal{M}$ and to different partition sets; *ii)* as show in Figure 8b, there is no cycle among the partition sets.

We now prove that in such circumstances the original DAI-graph is orientable if and only if each partition set, considered as a DAI-graph, is orientable. 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.

By $\mathcal{D}_\mathcal{M}[S_i]$, we denote the restriction of $\mathcal{D}_\mathcal{M}$ to the node set S_i.

Lemma 9. *Let $\mathcal{D}_\mathcal{M} = \langle V, A, X, E\rangle$ be a DAI-graph, and let $P = \{S_1, \ldots, S_n\}$ be a proper and acyclic partition of $\mathcal{D}_\mathcal{M}$. Then $\mathcal{D}_\mathcal{M}$ is orientable iff for all parts $S_i \in P$, the subgraph $\mathcal{D}_\mathcal{M}[S_i]$ is orientable.*

Proof. First, suppose there is a partition set $S_i \in P$ such that $\mathcal{D}_\mathcal{M}[S_i]$ is not orientable. Since $\mathcal{D}_\mathcal{M}[S_i]$ is a subgraph of $\mathcal{D}_\mathcal{M}$, all nodes, arcs, and disjunctive edges of $\mathcal{D}_\mathcal{M}[S_i]$ are contained in $\mathcal{D}_\mathcal{M}$. Then $\mathcal{D}_\mathcal{M}$ is not orientable either.

Next, suppose that for each $S_i \in P$, the subgraph $\mathcal{D}_\mathcal{M}[S_i]$ is orientable. Then $\mathcal{D}_\mathcal{M}[S_i]$ has an acyclic orientation $\mathcal{D}_i = \langle S_i, A_i\rangle$. We show how to construct an acyclic orientation of $\mathcal{D}_\mathcal{M}$. First, note that $\mathcal{D}_\mathcal{M}$ is itself acyclic, because P is acyclic and all $\mathcal{D}_i$ are acyclic, and each arc in A is either contained in some subgraph $\mathcal{D}_i$ or is between nodes of different subgraphs. Therefore, we must orient all disjunctive edges in E without introducing cycles. For edges entirely contained in a subgraph, the orientations are already contained in the corresponding subgraphs. Therefore, we can orient these edges in $\mathcal{D}_\mathcal{M}$ by removing them from E and adding all arcs from all subgraphs $\mathcal{D}_i$. Because P is proper, each exclusive group is fully contained in a single subgraph. Thus, we can find an orientation of the edges between different subgraphs by topologically ordering the subgraphs according to the arcs between them, i.e., by topologically ordering the partition sets in P. Each edge crossing partition sets 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 $\mathcal{D}_\mathcal{M}$. Hence, $\mathcal{D}_\mathcal{M}$ is orientable. □

Obviously, for every graph there is always a proper partition that is acyclic, namely the trivial one consisting of one set, the set of all nodes. To achieve a maximal reduction of the problem size, we must choose a proper partition with partition sets as small as possible. Such a partition can be found as follows. 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. (In the worst case, it may be that there is only one of them, that is, we only find the trivial partition.) After that we drop the auxiliary arcs. This construction ensures that *i)* each exclusive group is entirely contained in some 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 subgraphs $\mathcal{D}_1, \dots, \mathcal{D}_n$
        if NDSAT($\mathcal{D}_i$) = true for all $i \in \{1, \dots, n\}$ 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.

5.4. *Satisfiability Checking Evaluation*

We implemented the algorithm in Java and ran three series of experiments, *i)* to compare the running time of our algorithm with the one of a state-of-the-art model checker; *ii)* to identify to which extent the main optimization heuristics of the algorithm contribute to its performance; *iii)* to evaluate its scalability. All the experiments described in this section were run on the same machine, a Windows desktop PC with eight core Intel i7-4770 of 3.40 GHz and 8GB of RAM.

NuSMV [33] is a state-of-the-art model checker. We considered four models to compare it with our algorithm:

Satisfiable (Sat). A satisfiable hotel process model similar to the one in Figure 5.

With Cycle (Cycle). A variation of the consistent model such that the corresponding DAI-graph has a cycle.

Non-Orientable (N-Orient.). A variation of the consistent model such that the corresponding DAI-graph has no cycles but is not orientable, i.e., there is no orientation of the undirected edges such that the resulting graph has no cycles.

	Model		DAI-graph			NuSMV	SAT
Model	**Tasks**	**Dep.**	**Nodes**	**Arcs**	**Edges**	**Time**	**Time (ms)**
Sat	8	9	236	9415	524	2' 35"	27
Cycle	8	9	236	10003	521	> 1h	5
N-Orient.	12	14	244	9435	574	> 1h	10
Big	12	14	424	15131	1740	> 1h	23

Table 3: Comparison of NuSMV and our algorithm SAT

Non-Orientable bigger (Big). Similar to the Non-Orientable model the DAI-graph has more nodes.

The results of the experiment are reported in Table 3. The model checker NuSMV with Bounded Model Checking took 2 min 35 sec on the satisfiable model **Sat**. On the other models we stopped the verification after one hour. These results are not surprising if we consider the way satisfiability is checked in NuSMV. To perform the check a model is translated into a transition system representing the possible executions given the tasks in the model. Dependencies from the flow part of the model, instead, are translated into an LTL formula φ, and its negation $\neg\varphi$ is checked on the transition system. When the negated formula is violated, this means that there is one path in the transition system that satisfies φ, which corresponds to a possible execution. If $\neg\varphi$ is valid it means that there is no path satisfying φ, meaning that there is no execution satisfying all dependencies in the process model. In other words, the model is not satisfiable. It is then possible to see that if the model is satisfiable, NuSMV can stop its execution at the first violation found. Otherwise, it has to check all possible paths. This explains the results of the experiment.

As a second experiment we wanted to understand how much the observations described in Section 5.3 improved the verification. To this aim, we ran the experiment on the four models described before, by enabling/disabling the three optimization heuristics of *i)* Ignoring simple edges (**ISE**); *ii)* Resolving (**R**); and *iii)* Partitioning (**P**). The experiments were repeated three times and then the average time was computed and reported in Table 4, which shows the average time needed to translate a model into a DAI-graph, the average time needed for the check and the average overall time. The table shows that the combination of the three strategies improves the performance of the check. The verification is slower only in the **Cycle** case but the slow down is of a few milliseconds, thus negligible, especially considering that consistency checking need not be performed in real time.

Finally, we wanted to understand how the performance of the algorithm depends on the size of the graph. There are two ways of increasing the size of the graph, that is by increasing the number of locations in the model or by increasing the number of tasks. For convenience, we decided to increase the number of tasks. Specifically, we started from the non-orientable (**N-Orient.**) variant and in the experiment that we performed we copied it several times. We started from replicating the model 5 times and reached a total of 80 copies. To connect the copies, with a given probability we add a basic precedence constraint at global scope from some task of a model to some task of the other model. The choice of the tasks is random. To make sure we did not introduce cycles (which would simplify the check) we defined an order on the copies and added the precedences following that order. Finally, to make sure that the number of nodes in the graph was increased, we renamed the activities with new names in each copy. We chose this variant because it is the most challenging for the algorithm: *i)* the inconsistency is not due to a cycle, *ii)* the algorithm has to find a partition and non-deterministically chose an orientation, *iii)* the orientation, however, does not exist because the model is not orientable.

The results are reported in Table 5 and shown in Figure 9. On a model of 180 tasks, which we believe represents an average real case scenario, the performances are still acceptable (around 4 seconds). The

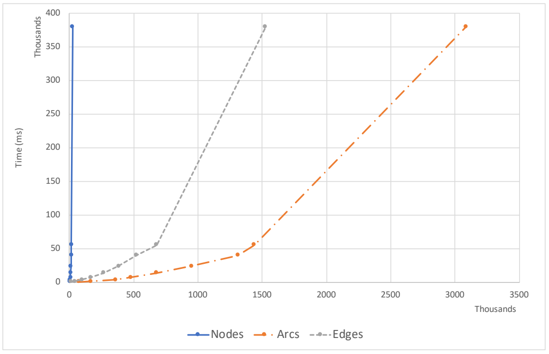
Model	ISE	R	P	Translation	Check	Total
Sat				2 ms	27,981 ms	27,984 ms
Sat	✓			2 ms	6,546 ms	6,548 ms
Sat		✓		5 ms	181 ms	187 ms
Sat	✓	✓		2 ms	15 ms	17 ms
Sat			✓	4 ms	16 ms	21 ms
Sat	✓		✓	2 ms	8 ms	11 ms
Sat		✓	✓	5 ms	16 ms	21 ms
Sat	✓	✓	✓	7 ms	19 ms	27 ms
Cycle				2 ms	<1 ms	3 ms
Cycle	✓			2 ms	1 ms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

Table 4: Experimental results on the four variants of the hotel process model, comparing the use of the strategies: Ignoring Simple Edges (ISE), Resolving (R), and Partitioning (P). Durations are rounded down to whole milliseconds.

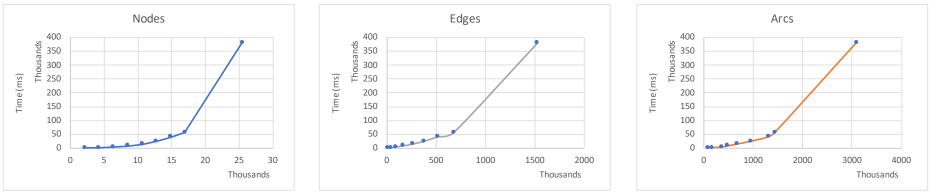
Model		DAI-graph			Alg.
Tasks	Dep.	Nodes	Arcs	Edges	Time (ms)
60	75	2,120	76,103	10,635	598
120	173	4,240	168,681	42,470	1,189
180	296	6,360	361,969	95,505	3,682
240	452	8,480	478,701	169,740	7,513
300	623	10,600	674,584	265,175	14,199
360	822	12,720	948,099	381,810	24,223
420	1,044	14,840	1,309,129	519,645	40,359
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 5: Experimental results to test the behavior of the algorithm changing the size of the model. (OOM Out-of-Memory)

algorithm 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, but we can expect that for inter-company process models this limit will never be reached. In these model, indeed, the level of detail in specifying the tasks and the items should be enough to represent the synchronization between the different companies present on-site, rather than defining step-by-step the intra-company process. The graph in Figure 9 reports the relations between the three variables in a graph, i.e., the number of nodes, edges and arcs, and the time needed to perform the check. As can be seen the relation seems to be exponential and grows very fast in the number of nodes.

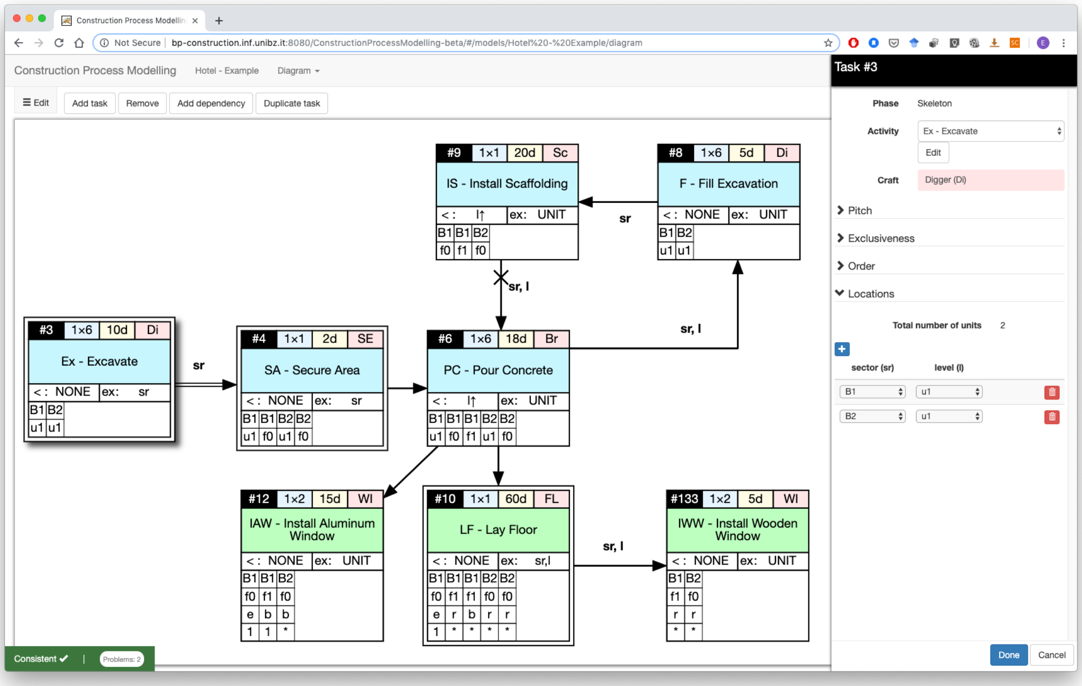


(a) Growing of time depending on nodes, arcs and edges.



(b) Individual graphs showing the growing of time with respect to the growing of nodes, arcs and edges.

Figure 9: Relation between number of nodes, arcs and edges with the time needed for the algorithm to check satisfiability.



Task #3

Phase: Skeleton
Activity: Ex - Excavate
Craft: Digger (Di)
Pitch
Exclusiveness
Order
Locations
Total number of units: 2

sector (sr)	level (l)
B1	u1
B2	u1

Figure 10: Flow Part.

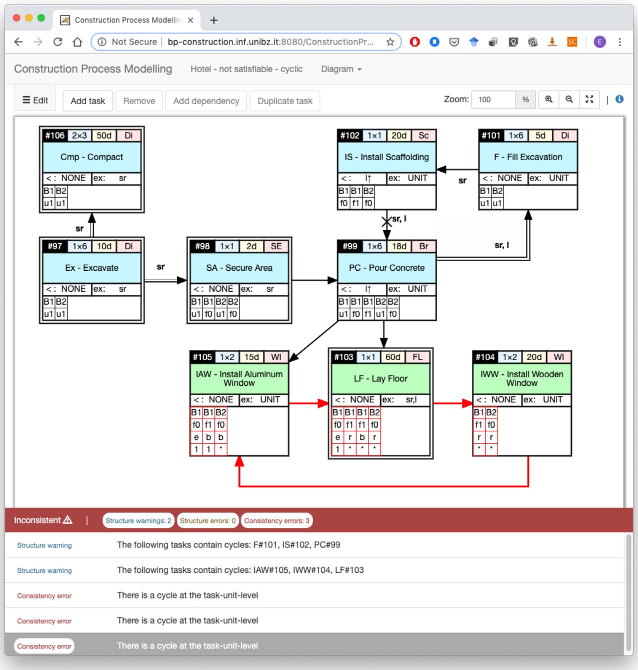
6. CoPMod: IT Support for Construction Process Modeling

In order to acquire feedback from construction companies and to explain our approach for construction process modeling to potentially interested people, we developed a proof-of-concept tool called CoPMod: Construction Process Modeling. It supports the process modeling and the satisfiability checking as described in this paper and can be seen at `copmod.inf.unibz.it` (a screencast is available at `copmod.inf.unibz.it/copmod.html`).

To start using the tool, first one has to create a new project. This is possible by creating a project *i)* from scratch; *ii)* as a clone of an existing project; or *iii)* by importing a specification of a project which must be provided as a JSON file. A project is conceptually organized into a *configuration* and a *flow* part. In the configuration part one has to specify information such as the construction perspectives; the locations, in terms of attributes, their range of values, the item structure and the item values (see Section 3.1); the crafts; and the foreseen activities assigned to a construction perspective. Currently this is possible by uploading a JSON file containing the specification. It will later be supported by a proper graphical interface.

The flow part supports the companies, orchestrated by a moderator familiar with the tool and the language (likely the project manager responsible for the project) in drawing a diagram containing tasks and dependencies on their execution. Figure 10 reports the flow part diagram for the hotel example.

By relying on a configuration part defining the building, the set of crafts and of activities, the tool limits the introduction of insertion errors (such as typos). Additionally, CoPMod checks for *structure warnings*, *structure errors* and *consistency errors*. Structure warnings show parts of the diagram that potentially can be rewritten in a better way, such as a location repeated in a task box, or cycles that do not create inconsistencies but that may mean that some constraint is redundant. Structure errors indicate that something is missing in



Inconsistent | Structure warnings: 2 | Structure errors: 0 | Consistency errors: 3

```
Structure warning    The following tasks contain cycles: F#101, IS#102, PC#99
Structure warning    The following tasks contain cycles: IAW#105, IWW#104, LF#103
Consistency error    There is a cycle at the task-unit-level
Consistency error    There is a cycle at the task-unit-level
Consistency error    There is a cycle at the task-unit-level
```

Figure 11: Warnings and Consistency Checks.

the diagram (e.g., an activity is not associated to a construction perspective). More interestingly, the tool implements the *consistency check*. When the tool visualizes a consistency error, it means that the model cannot be satisfied by any execution. In Figure 11 several warnings and errors are signaled. For instance, the loop between Install Aluminum Window, Lay Floor and Install Wooden Window cannot be satisfied by any execution. The loop between Pour Concrete, Fill Excavation and Install Scaffolding, instead, is satisfiable and thus is not highlighted as a consistency error but as a structure warning.

The tool has been implemented as a RESTful web application, so that it can be accessed by the interested companies without having to install or setup anything. Several libraries and technologies have been used for the implementation, among which the main ones are Java for the back-end, JavaScript and AngularJS for the front-end and Spring Data REST to generate the REST API. MySQL was used as DBMS.

7. Related Work

In this paper we presented inter-disciplinary work that involves the computer science and construction research areas. Accordingly, in Section 7.1 we relate work that focuses on process modeling and formal verification from the computer science point of view, while in Section 7.2, we discuss current and emerging approaches for process management in construction.

7.1. Process Modeling in Computer Science

In computer science, (Business) Process Management is a research branch that addresses several aspects related to processes, such as modeling, discovery, verification, optimization and so on. The work presented in this paper focuses on process modeling and on an efficient way to verify satisfiability of a model.

Concerning process modeling, one of the main distinctions that can be found in the literature is between *procedural* and *declarative* approaches [25]. In the first case, a model is defined by representing all and only the executions that are allowed. Everything that is not specified is forbidden. Declarative approaches, instead, represent what is required or forbidden for an execution to be compliant with a model. All executions that satisfy the declarative requirements are allowed but executions are not represented explicitly. Concerning the procedural approaches, one of the best known is the Business Process Modeling Notation (BPMN) that we already discussed in Section 2.2. BPMN has been widely investigated from several point of views. For instance, a number of tools to support modeling and execution of BPMN processes have been proposed among which Bizagi [34], Bonita Software [35], Camunda [36], and Signavio [37]. Several studies focused on the verification of formal properties over a model [38]. One of the adopted techniques is to use model checking, as in [39], where the author proposes a translation of a BPMN process into a NuSMV model. By using the model checker it is then possible to verify properties expressed as LTL or CTL formulas. In [40] compliance of a process model is checked against a number of patterns that can be expressed in *BPMN-Q*, a graphical language for querying BPMN process models [41]. Also in this approach model checking techniques are applied: the patterns are translated into temporal formulas and then are checked against a model. However, to reduce the state-space problem of model checking, the authors propose a technique where only the relevant parts of a model, w.r.t. a path to check, are considered for the verification.

All of these approaches aim at modeling and checking properties that are mainly related to the control flow. BPMN, indeed, focuses mainly on how to capture the possible executions, while the data on which activities are to be executed is considered in a limited way [4, 19, 20]. As discussed in Section 2.2, the specification of the locations on which activities must be performed is possible adopting an ad-hoc solution where a BPMN activity is actually a pair of construction activity and location (see Figures 2 and 3). This, however, makes a model more complex (in terms of size and dependencies), difficult to understand, use and maintain. Therefore, with respect to the requirements identified in Section 2.1, BPMN fails in representing in an adequate way the elements that are needed for construction process models and does not support flexibility and adaptability of a model. Some studies in the literature also show that BPMN specifications can be ambiguous [19, 20], which is mainly due to the generality of the approach, which makes it possible to capture the same behaviour in different ways.

Of course, BPMN is not the only procedural approach for process modeling. Petri Nets (PN), UML Diagrams and Session Types have been used for several purposes among which process modeling and property verification. For instance, in [42] the authors investigate how to enforce soundness and termination by construction, on a model defined in PN. Session Types in [43] have been defined in such a way as to guarantee liveness for non-terminating processes. These approaches, however, are general purpose and procedural, suffering from the problems that we already discussed.

Declarative approaches are an alternative to procedural ones. In general, declarative approaches better suit dynamic and collaborative settings, where flexibility of process models is a desired characteristic [44, 23, 14, 22, 45]. Among the best known declarative approaches are Declare [27], the case handling paradigm [46, 47], and DCR Graphs [45, 48].

Declare is graphical language with an LTL_f formal semantics. The graphical representation was introduced aiming at supporting the definition and comprehension of process models, aspects that might be difficult to achieve in declarative approaches and which are supposed to be better realized in procedural ones. As discussed in Section 2.2, Declare suffers from some of these drawbacks, which makes it not suited for construction process modeling. Specifically, similarly to BPMN, also Declare focuses on the control flow, making it complicated to represent the items on which activities have to be performed. As in BPMN, this can be done by representing each activity as an activity-location pair, making, however, the process model more complicated. Additionally, Declare does not support the representation of alternative constraints, which are needed to capture exclusivity constraints. If an activity has an exclusivity constraint on a set of locations, then any other activity to be performed on one such location can be performed either before or after the activity is executed on all the locations, but not in between. In Declare it is possible to express alternative between activities (like the not-coexistence constraint or branching constraints as in [49]) but not between patterns. These are the main reasons why CoPModL does not rely on Declare. Note, indeed, that CoPModL is similar to Declare in the declarative nature, the naming of the constraints and their intuitive interpretation, but the formal semantics is different.

Different aspects of Declare have been investigated in the literature, such as process mining, formal verification of properties, and extensions of the language to capture more expressive execution conditions. In general, a shortcoming that is shared by all of these approaches is that, since they rely on Declare, they suffer from the problems discussed above. Let us discuss some of them more in detail.

In the context of process mining, the authors in [24, 50] focuses on two properties of Declare process models that are automatically extracted from process logs, namely consistency and redundancy. They consider a subset of the Declare language and introduce techniques based on automata-products to check the two properties and to propose a new model without inconsistencies and redundancies. The possibility to suggest an alternative process model, similar to the original one, is very interesting but relies on the assumption that each constraint retains information such as the support, confidence and interest factor, that can be obtained from the process log during the discovery phase. In our setting, we cannot count on the availability of process logs and we focus on a simpler problem, that is to check whether a model is consistent or not. Moreover, the consistency and redundancy checks that they introduced can be defined on the Declare constraint templates. In our case, considering constraint templates would not be enough, since also locations and the scope of a template play an active role in process satisfiability. Finally, the semantics provided in [24, 50] is given by regular expressions, which naturally fits with the purpose of the paper, i.e., automata-based verification of properties. In our case, we aim at specifying properties that a linear execution should satisfy. For this purpose, a semantics as LTL_f represents a natural choice.

Multi-Perspective Declare (MP-Declare) has been introduced as an extension of Declare to consider also data and temporal perspectives beside the control flow one. In this proposal, Declare templates are extended with the possibility to specify an *activation condition*, which is related to the activating activity and specifies which condition should be met, besides the occurrence of the activity, in order for the constraint to be activated; a *target condition*, expressing a condition to be met for the constraint to be fulfilled and concerning the target activity of the constraint; a *correlation condition* which is a condition that relates aspects of both activities and is evaluated when the target is achieved. In [23] the authors investigate the problem of checking conformance, that is verifying that a process instance complies with an MP-Declare process model. The conditions they express are over data elements, which can be event attributes (i.e., data produced by an activity of the instance) or over case attributes (i.e., data concerning the whole process instance). The authors in [13] focus on the execution of MP-Declare processes. In particular, they are interested in checking whether, given a partial process instance, it can be completed without violating any constraint. This property is similar to the conformance check in [23], but it is performed over partial traces. In both cases, the two properties are different from satisfiability, which we consider in our work, where no trace is given but the existence of one satisfying all the constraints must be checked. Another line of research related to Declare is about mining of processes extended with data. In particular, [44] focuses on MP-Declare processes mining, while [14] focuses on similar processes but where only activation conditions are considered.

From the literature on MP-Declare two aspects emerge clearly: *i)* the usefulness of declarative languages (at least in dynamic domains), and *ii)* the need of extending the control flow specification of Declare with conditions defined on (process instance) data. Our approach is very much in line with both aspects, given the flexibility and adaptability requirements of construction processes and the need of expressing locations and location-based dependencies. The language and the conditions defined in MP-Declare are able to capture a variety of constraints, which however are different from the items on which a CoPModL activity has to be executed and the scope of a dependency. MP-Declare conditions, indeed, aim at capturing when a constraint is activated and, if activated, what conditions must be satisfied not to violate it. In CoPModL, locations and the scope are part of the model definition, specifying exactly what needs to be done, where it needs to be done, and how the different tasks are related one to another. However, CoPModL constraints do not have any "conditional" nature on the activation of the constraint.

Dynamic Condition Response (DCR) Graphs [45, 48] in their graphical format resemble the Declare-language. A graph consists of events and relations among them. Relations can be of four kinds: dynamic inclusion, dynamic exclusion, condition and response. The work in [48] is considering the case in which a process model is continuously adapted to achieve specific sub-goals. In this settings some properties need to be ensured all the time. Therefore, they automatically check the absence of deadlock and livelock by applying model checking techniques. The work in [45] extends DCR Graphs by introducing nesting of graphs. This nesting is defined in such a way to ensure consistency of the model. Consistency holds if, when flattening a nested graph, there are no events that are both excluded (something should not occur) and included (something must occur). In this case, consistency is enforced, rather than checked.

Another declarative approach is the case handling paradigm [46, 51] which introduces a shift from a flow perspective to a case perspective. A *case* can be seen as a product to be realized and the idea is that a process must be represented in a flexible way, thus the choice of being declarative, and the workers need to have at any time all information about a case. Accordingly, a case is defined with a structure and a current state, which in turn is represented as a collection of data objects. Therefore, the current state of a case is represented by the presence of data objects. This represents a shift of perspective where, besides the control flow also the data-flow is considered. Data objects are linked to activities and may be mandatory, i.e., needed to complete the activity, or restricted, that is, the data object is needed to complete the activity and the process cannot progress with other activities. Synchronization between several instances and activities is achieved via *data objects*. Additionally, activities may be non-atomic and are assigned to roles which can have different capabilities such as execute an activity, redo it, undo it or skip it. In 2014 the OMG introduced the Case Management Model and Notation [47] (CMMN) standard, which is a graphical notation for case-based process modeling. In [1], the paradigm has been applied to construction, sharing with our work similar requirements (such as adaptability and flexibility).

However, the focus of [1] is on the "preparation of execution phase", where a number of documents, drawings and such like need to be orchestrated. Instead, the focus of our work is on the execution phase, where the kind of information to represent and the level of detail needed are different.

Some approaches in the literature try to consider different or additional perspectives, other than the control flow. The instance-spanning constraints [6] approach aims at dealing with constraints that relate several process instances of one or several process schemas. An example could be the need to express the synchronization between several instances before the process can progress further. In [6] the authors identify several properties for the constraints: *i) localization*, which defines whether a constraint should be enforced for a single instance, all instances of a process, instances of many processes, or instances of processes of different organizations; *ii)* the *span*, which considers whether a constraint that affects multiple tasks, concerns tasks of the same instance, of multi-instances, of multi-processes or of multi-organizations; *iii) dependency*, which concerns executions that depend on previous executions (e.g., some task can be executed only if a certain number of executions of another task have already occurred).

The work on Object-Centric Behavioural Constraints [5] also is concerned with expressing constraints over several process instances, focusing on cardinality constraints to relate flow and data constraints. Also in this approach, items are not represented explicitly, which in our approach plays a role in checking the consistency of a model. Also, representation of item-dependent relationships is not considered in this approach.

The approaches that we described up to know have been classified as *activity-centric* approaches, because the focus is mainly on the activities to be performed and their (temporal, ordering) relation. If they can represent data, this is considered only in a second place and usually in a limited way (e.g., BPMN). A different perspective is represented by the *data-centric* approaches, of which the *artifact-centric approach* [52] can be considered as one representative. The idea is to model a process by considering a number of entities (artifacts) that are relevant in the setting and for the purpose the model is made. Each artifact is characterized by an information model and a lifecycle model. The former stores business-relevant information and the latter specifies the possible evolution of the artifact over time. In [53, 54], the lifecycle is specified in terms of Guard-Stage-Milestone. Here, an artifact is organized into stages (which can be composed or atomic). A guard specifies a condition that makes the stage active, meaning that the tasks, in terms of which a stage is defined, can be performed. A task can be an assignment (for or from attributes in the information model), service invocations, human tasks or the creation of an artifact instance. A milestone represents a condition that closes the stage. When more than one milestone is specified, then the stage is closed when one is met. The process is then defined in terms of interactions between the business artifacts and their evolution.

Data-centric approaches represented a switch of perspective, from the control flow to the data (artifact) perspective. Accordingly, this kind of representation suits those situations in which there are a number of well-defined entities to be modeled, and the aim of the model is in capturing their evolution and interaction. In the construction setting that we discussed, the aim of the model is closer to the *activity*-centric approaches, aiming at specifying in the first place the activities and the locations where the companies have to perform some work, and then the relations among these. In the artifact centric approach, the perspective of activities and the order of their execution is hidden inside the specification of an artifact and its data model and stages. For this reason we believe that artifact-centric approaches are not the right solution to express in an effective way the coordination among several construction companies.

7.2. Process Management in Construction

In construction, the traditional and most adopted techniques are *activity-based* approaches, focusing mainly on the activities to be performed. The most famous ones are the Critical Path Method (CPM) [55], established in 1960 [9], and the Program Evaluation and Review Technique (PERT) [56]. These approaches rely on the definition of an activity network, where nodes specify the activities and their duration and arcs represent orderings between the activities. For such a network, one can compute a *critical path*, that is, a path in the network such that if one activity on the path is delayed, then the overall duration of the project increases. Instead, for non-critical paths there is a time buffer, such that a certain activity delay can be absorbed without increasing the overall duration of the project.

CPM and PERT are simple enough to be used also without IT support and this contributed to their initial success and wide adoption. However, these approaches have some limitations: mainly they do not account for the locations where activities are to be executed. Since locations are not considered, it is also not possible to express location-based relationships between the activities [9]. As a result, the process representation abstracts from important details, causing [57]: *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 updated during the execution, which would be necessary if one wanted to use them for process control [9].

Gantt charts are a graphical tool for scheduling in project management. Being graphical, these charts are intuitive and naturally support the visualization of when an activity is planned to be executed and of how long is its duration. By connecting the activities with arrows it is also possible to represent some dependencies among them. Similarly to the approaches described previously, also Gantt charts do not support naturally the definition of the locations. Practically, they have been used representing locations as activities and activities as sub-activities of the locations. This has some drawbacks, like the impossibility to explicitly represent location-based relationships and their scope. Moreover, a Gantt chart already represents a commitment to one particular schedule and does not constitute a more declarative process model that would specify only the requirements on the allowed/desired executions.

Such a declarative process model would allow for a more flexible approach: in case of a delay or unforeseen events any re-scheduling which satisfies the model leads to a possible schedule. In case only a schedule is provided, the requirements (such as tasks to be executed before/after other tasks) are not explicit, thus, it might not be clear how to perform the (re-)scheduling. The limitations of Gantt charts also apply to the IT-tools supporting the approach (such as Microsoft Project).

Orthogonal to activity-based approaches there are the *location-based* approaches where the main focus is on the locations where activities need to be performed. The *Line-of-Balance* (LOB) [58] method has been introduced in the first half of the twentieth century, with the aim of addressing repetitive constructions (that is, projects where some sub-process is repeated frequently, for instance at each floor). This method plots the planned activities on a chart, where the axes represent, respectively, the time and the quantity (of work) that can be delivered at a certain point in time. Activities are then represented by lines, showing how much of an activity can be achieved at a certain point in time. Accordingly, the slope of a curve represents the production line of the corresponding activity, which can be varied by varying the number of crews assigned to the activity: more crews means that the activity can progress faster, thus a higher quantity can be delivered in a certain amount of time. This approach has been appreciated because it easily supports an evaluation of the *workflow continuity* in performing the activities. Specifically, if the lines representing two activities are continuous and parallel then this indicates a continuous workflow, intuitively meaning that the two activities progress at the same pace and that one succeeds the other, location by location. If the two activities have different pace, then parallelism can be achieved by changing the number of crews working on the activity: fewer or more crews to progress more slowly or faster, respectively. In this way, the graphical representation supports a form of resource management.

It is not hard to see that this approach specifically suits repetitive tasks, that is tasks that must be repeated in several locations always taking the same time. Similar approaches are the *flowline* [59] and the *Location-Based Management System* [9] methods. The differences with LOB are that axes represent time and location (instead of quantities) and that the activities are represented with lines in a slightly different way. Location-based approaches thus focus on workflow continuity and on productivity. Similarly to some of the activity-based approaches, they do not have an explicit model with the requirements to be satisfied by an execution. Also in these approaches, indeed, a chart already represents a schedule and the commitment to established dates.

The *Building Information Modeling* (BIM) process was introduced more or less at the same time, however it become popular only recently. One of the main aims of BIM is to facilitate the exchange of digital information concerning the functional and the physical characteristics of a building [60]. The underlying idea is that different actors with complementary expertise (e.g., architects, engineers, foremen) can work on the same model and enrich it with discipline-specific information. As well as the tools supporting BIM, the approach is very powerful, but the abundance of information makes it also difficult to manage and maintain, and requires dedicated resources to do that [61], resources that not all companies are willing or able to afford (in particular small and medium-size enterprises). As a result, either BIM is not adopted, or it is adopted only in the initial phases of a project and then, given the effort required to maintain it, in many cases a model is not updated.

All these factors contribute to a resistance against IT-support in construction and result in a lower adoption of IT-tools in the industry, compared to others, such as manufacturing [62].

8. Conclusions and Future Work

This work presents an approach and the language CoPModL 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 efficient algorithm to check it. The algorithm has been implemented in CoPMod, a proof-of-concept tool that also supports the graphical definition of a process model [17].

The motivation for developing a formal approach for process modeling emerged in the application of non-formal models in real projects [7, 10], 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 will apply modeling and automatic scheduling to real construction projects in the context of the research project COCkPiT [8]. We are also investigating the integration with BIM-based tools (such as Autodesk® Revit or Archicad) so as to extract the geometries of the buildings and assign activities and quantities to be performed there. This would support both process modeling and scheduling.

Acknowledgments. This work was supported by the projects MoMaPC, financed by the Free University of Bozen-Bolzano, and by the research project “COCkPiT – Collaborative Construction Process Management – FESR1008”, funded by the European Regional Development Fund (ERDF) of the Autonomous Province of Bolzano-South Tyrol.

References

[1] W. van der Aalst, M. Stoffele, J. Wamelink, Case Handling in Construction, Automation in Construction 12 (3) (2003) 303–320.

[2] M. Dumas, 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, pp. XV–XVI.

[3] U. Frank, Multilevel Modeling - Toward a New Paradigm of Conceptual Modeling and Information Systems Design, Business & Information Systems Engineering 6 (6).

[4] D. Calvanese, G. De Giacomo, M. Montali, Foundations of Data-Aware Process Analysis: a Database Theory Perspective, in: PODS, ACM, 2013, pp. 1–12.

[5] W. M. P. van der Aalst, A. Artale, M. Montali, S. Tritini, Object-Centric Behavioral Constraints: Integrating Data and Declarative Process Modelling, in: Description Logics, 2017, pp. 1–12.

[6] M. Leitner, J. Mangler, S. Rinderle-Ma, Definition and Enactment of Instance-Spanning Process Constraints, in: Web Information Systems Engineering, LNCS 7651, 2012, pp. 652–658.

[7] Build4Future, `www.fraunhofer.it/en/focus/projects/build4future.html`.

[8] COCkPiT: Collaborative Construction Process Management, `www.cockpit-project.com/`.

[9] R. Kenley, O. Seppänen, Location-Based Management for Construction: Planning, Scheduling and Control, Routledge, 2006.

[10] E. Marengo, P. Dallasega, M. Montali, W. Nutt, M. Reifer, Process Management in Construction: Expansion of the Bolzano Hospital, in: Business Process Management Cases, Springer, 2018, pp. 257–274.

[11] G. De Giacomo, M. Y. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: 23rd International Joint Conference on Artificial Intelligence (IJCAI), 2013, pp. 854–860.

[12] S. Schönig, M. Zeising, S. Jablonski, Towards Location-Aware Declarative Business Process Management, in: Business Information Systems Workshops, Vol. 183 of LNBIP, Springer, 2014, pp. 40–51.

[13] L. Ackermann, S. Schönig, S. Petter, N. Schützenmeier, S. Jablonski, Execution of Multi-perspective Declarative Process Models, in: On the Move to Meaningful Internet Systems. (OTM), Vol. 11230 of LNCS, Springer, 2018, pp. 154–172.

[14] F. M. Maggi, M. Dumas, L. García-Bañuelos, M. Montali, Discovering Data-Aware Declarative Process Models from Event Logs, in: Business Process Management - 11th International Conference, BPM, Vol. 8094 of LNCS, Springer, 2013, pp. 81–96.

[15] Y. Lu, X. Xu, J. Xu, Development of a Hybrid Manufacturing Cloud, Journal of Manufacturing Systems 33 (4).

[16] E. Marengo, W. Nutt, M. Perktold, Construction Process Modeling: Representing Activities, Items and Their Interplay, in: Business Process Management BPM 2018, LNCS 11080, Springer, 2018, pp. 48–65, Best Paper Award.

[17] E. Marengo, W. Nutt, M. Perktold, CoPMod: Support for Construction Process Modeling, in: Proceedings of the Dissertation Award, Demonstration, and Industrial Track at BPM 2018, CEUR 2196, CEUR-WS.org, 2018, pp. 61–65.

[18] M. Dumas, M. L. Rosa, J. Mendling, H. A. Reijers, Fundamentals of Business Process Management, Second Edition, Springer, 2018.

[19] E. Börger, Approaches to Modeling Business Processes: a Critical Analysis of BPMN,Workflow Patterns and YAWL, Software & Systems Modeling 11 (3) (2012) 305–318.

[20] P. Wohed, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, N. Russell, On the Suitability of BPMN for Business Process Modelling, in: Business Process Management, 4th International Conference, Vol. 4102 of LNCS, Springer, 2006, pp. 161–176.

[21] O. Savkovic, E. Marengo, W. Nutt, Query Stability in Monotonic Data-Aware Business Processes, in: ICDT, Vol. 48 of LIPIcs, 2016, pp. 16:1–16:18.

[22] M. Zeising, S. Schönig, S. Jablonski, Towards a Common Platform for the Support of Routine and Agile Business Processes, in: 10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, ICST / IEEE, 2014, pp. 94–103.

[23] A. Burattin, F. M. Maggi, A. Sperduti, Conformance Checking Based on Multi-Perspective Declarative Process Models, Expert Syst. Appl. 65 (2016) 194–211.

[24] C. Di Ciccio, F. M. Maggi, M. Montali, J. Mendling, Resolving Inconsistencies and Redundancies in Declarative Process Models, Information Systems 64 (2017) 425–446.

[25] D. Fahland, D. Lübke, J. Mendling, H. A. Reijers, B. Weber, M. Weidlich, S. Zugal, Declarative Versus Imperative Process Modeling Languages: The Issue of Understandability, in: 10th International Workshop on Enterprise, Business-Process and Information Systems Modeling, Vol. 29 of LNBIP, Springer, 2009, pp. 353–366.

[26] W. M. P. van der Aalst, M. Pesic, H. Schonenberg, Declarative Workflows: Balancing Between Flexibility and Support, Computer Science-R&D 23 (2).

[27] W. M. P. van der Aalst, M. Pesic, DecSerFlow: Towards a Truly Declarative Service Flow Language, in: Web Services and Formal Methods, Springer, 2006, pp. 1–23.

[28] M. Pesic, H. Schonenberg, W. M. P. van der Aalst, DECLARE: Full Support for Loosely-Structured Processes, in: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), IEEE Computer Society, 2007, pp. 287–300.

[29] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, (FOCS), 1977, pp. 46–57.

[30] M. Perktold, E. Marengo, W. Nutt, CoPMod: Construction Process Modeling (Proof-of-concept tool), `http://copmod.inf.unibz.it`.

[31] M. Perktold, Processes in Construction: Modeling and Consistency Checking, Master's thesis, Free University of Bozen-Bolzano, `http://pro.unibz.it/library/thesis/00012899S_33593.pdf` (2017).

[32] P. Fortemps, M. Hapke, On the Disjunctive Graph for Project Scheduling, Foundations of Computing and Decision Sciences 22.

[33] Nusmv, `http://nusmv.fbk.eu/` (June 2019).

[34] Bizagi, `https://www.bizagi.com/` (June 2019).

[35] Bonitasoft, `https://www.bonitasoft.com/` (June 2019).

[36] Camunda, `https://camunda.com/bpmn/tool/` (June 2019).

[37] Signavio, `https://www.signavio.com/` (June 2019).

[38] H. Groefsema, D. Bucur, A Survey of Formal Business Process Verification: From Soundness to Variability, in: International Symposium on Business Modeling and Software Design, 2013, pp. 198–203.

[39] V. S. W. Lam, Formal Analysis of BPMN Models: a NuSMV-Based Approach, International Journal of Software Engineering and Knowledge Engineering 20 (7) (2010) 987–1023.

[40] A. Awad, G. Decker, M. Weske, Efficient Compliance Checking Using BPMN-Q and Temporal Logic, in: Business Process Management, 6th International Conference, Vol. 5240 of LNCS, Springer, 2008, pp. 326–341.

[41] A. Awad, BPMN-Q: A Language to Query Business Processes, in: Proceedings of EMISA'07, 2007, pp. 115–128.

[42] K. M. van Hee, N. Sidorova, J. M. E. M. van der Werf, Business Process Modeling Using Petri Nets, Trans. Petri Nets and Other Models of Concurrency 7480 (2013) 116–161.

[43] S. Debois, T. T. Hildebrandt, T. Slaats, N. Yoshida, Type Checking Liveness for Collaborative Processes with Bounded and Unbounded Recursion, in: Formal Techniques for Distributed Objects, Components, and Systems, Vol. 8461 of LNCS, Springer, 2014, pp. 1–16.

[44] S. Schönig, C. Di Ciccio, F. Maria Maggi, J. Mendling, Discovery of Multi-perspective Declarative Process Models, in: Service-Oriented Computing - 14th International Conference, ICSOC, Vol. 9936 of LNCS, Springer, 2016, pp. 87–103.

[45] T. T. Hildebrandt, R. R. Mukkamala, T. Slaats, Nested Dynamic Condition Response Graphs, in: Fundamentals of Software Engineering - 4th IPM International Conference, Vol. 7141 of LNCS, Springer, 2011, pp. 343–350.

[46] W. M. P. van der Aalst, M. Weske, D. Grünbauer, Case Handling: a New Paradigm for Business Process Support, Data Knowl. Eng. 53 (2) (2005) 129–162.

[47] Case Management Model and Notation, `https://www.omg.org/cmmn/` (June 2019).

[48] R. R. Mukkamala, T. T. Hildebrandt, T. Slaats, Towards Trustworthy Adaptive Case Management with Dynamic Condition Response Graphs, in: 17th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2013, IEEE Computer Society, 2013, pp. 127–136.

[49] M. Pesic, W. M. P. van der Aalst, A Declarative Approach for Flexible Business Processes Management, in: Business Process Management Workshops, Vol. 4103 of LNCS, Springer, 2006, pp. 169–180.

[50] C. Di Ciccio, F. M. Maggi, M. Montali, J. Mendling, Ensuring Model Consistency in Declarative Process Discovery, in: Business Process Management - 13th International Conference, BPM 2015, Vol. 9253 of LNCS, Springer, 2015, pp. 144–159.

[51] W. M. P. van der Aalst, P. J. S. Berens, Beyond Workflow Management: Product-driven Case Handling, in: Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work, ACM, 2001, pp. 42–51.

[52] R. Hull, Artifact-Centric Business Process Models: Brief Survey of Research Results and Challenges, in: On the Move to Meaningful Internet Systems: OTM 2008, Vol. 5332 of LNCS, Springer, 2008, pp. 1152–1163.

[53] R. Vaculín, R. Hull, T. Heath, C. Cochran, A. Nigam, P. Sukaviriya, Declarative Business Artifact Centric Modeling of Decision and Knowledge Intensive Business Processes, in: Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference, IEEE Computer Society, 2011, pp. 151–160.

[54] R. Hull, E. Damaggio, R. De Masellis, F. Fournier, M. Gupta, F. F. T. Heath III, S. Hobson, M. H. Linehan, S. Maradugu, A. Nigam, P. N. Sukaviriya, R. Vaculín, Business Artifacts with Guard-Stage-Milestone Lifecycles: Managing Artifact Interactions with Conditions and Events, in: Proceedings of the Fifth ACM International Conference on Distributed Event-Based Systems, ACM, 2011, pp. 51–62.

[55] J. E. Kelley Jr, M. R. Walker, Critical-path Planning and Scheduling, in: Eastern Joint IRE-AIEE-ACM Computer Conference, IRE-AIEE-ACM '59 (Eastern), ACM, 1959, pp. 160–173.

[56] U. Navy, Program Evaluation Research Task, Summary Report Phase 1, AD-735 902.

[57] A. Shankar, K. Varghese, 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, pp. 1447–1455.

[58] P. Lumsden, The Line-of-Balance Method, Pergamon Press Limited, 1968.

[59] W. E. Mohr, Project Management and Control:(in the Building Industry), Department of Architecture and Building, University of Melbourne, 1978.

[60] B. Hardin, D. McCool, BIM and Construction Management: Proven Tools, Methods, and Workflows, John Wiley & Sons, 2015.

[61] P. Forsythe, S. Sankaran, C. Biesenthal, How Far Can BIM Reduce Information Asymmetry in the Australian Construction Context?, Project Management Journal 46 (3).

[62] KPMG International: Building a Technology Advantage. Harnessing the Potential of Technology to Improve the Performance of Major Projects, Global Construction Survey (2016).