Software and Systems Modeling (2021) 20:1379–1401
https://doi.org/10.1007/s10270-020-00857-8

SPECIAL SECTION PAPER

Language-independent look-ahead for checking multi-perspective declarative process models

Martin Käppel[1] · Lars Ackermann[1] · Stefan Schönig[2] · Stefan Jablonski[1]

Received: 15 December 2019 / Revised: 26 November 2020 / Accepted: 22 December 2020 / Published online: 9 January 2021

Abstract
Declarative process modelling languages focus on describing a process by restrictions over the behaviour, which must be satisfied throughout the whole process execution. Hence, they are well suited for modelling knowledge-intensive processes with many decision points. However, such models can be hard to read and understand, which affect the modelling and maintenance of the process models tremendously as well as their execution. When executing such declarative (multi-perspective) process models, it may happen that the execution of activities or the change of data values may result in the non-executability of crucial activities. Hence, it would be beneficial to know all consequences of decisions to give recommendations to the process participants. A look-ahead attempts to predict the effects of executing an activity towards possible consequences within an a priori defined time window. The prediction is based on the current state of the process execution, the intended next event and the underlying process model. While execution engines for single-perspective imperative process models already implement such functionality, execution approaches, for multi-perspective declarative process models that involve constraints on data and resources, are less mature. In this paper, we introduce a simulation-based look-ahead approach for multi-perspective declarative process models. This approach transforms the problem of a context-aware process simulation into a SAT problem, by translating a declarative multi-perspective process model and the current state of a process execution into a specification of the logic language Alloy. Via a SAT solver, process trajectories are generated that either satisfy or violate this specification. The simulated process trajectories are used to derive consequences and effects of certain decisions at any time of process execution. We evaluate our approach by means of three examples and give some advice for further optimizations.

Keywords Declarative process models · Multi-perspective · Look-ahead · Model checking · Predictive business process monitoring · SAT solving

1 Introduction

In the last years, business process management (BPM) has gained massive importance for enterprises of all industries and sizes in order to increase their efficiency, to reduce costs and to be compliant with legal regulations [15]. Two opposing classes of business processes can be identified: *routine processes* and *knowledge-intensive processes*[1] [16,17].

Imperative processes are considered to be stable, predictable and determined [16,29]. They encompass less variants and their execution barely depends on the knowledge and decisions of (human) process participants [29]. Hence, at any time the exact control flow is known, as well as the dependencies between different process perspectives (e.g. process participants or data in form of variables or formulars). In contrast, the exact control flow of declarative processes is not given a priori and their execution strongly depends on

Communicated by Jens Gulden and Rainer Schmidt.

✉ Martin Käppel
martin.kaeppel@uni-bayreuth.de

Lars Ackermann
lars.ackermann@uni-bayreuth.de

Stefan Schönig
stefan.schoenig@ur.de

Stefan Jablonski
stefan.jablonski@uni-bayreuth.de

[1] University of Bayreuth, Bayreuth, Germany

[2] University of Regensburg, Regensburg, Germany

[1] Sometimes knowledge-intensive processes also called flexible or decision-intensive processes.

the knowledge and the decisions of the (human) process participants [29].

This distinction is also reflected in the associated process modelling languages. The so-called imperative process modelling paradigm includes languages such as Petri nets [41] and BPMN [20], focuses on describing all execution paths explicitly in a graph-based structure. Hence, the languages are well suited to structured processes with little variation. On the other hand, the so-called declarative paradigm, including languages such as Declare [28], Multi-Perspective-Declare (MP-Declare) [8], DCR graphs [22], GSM [23] and the Declarative Process Intermediate Language (DPIL) [31,44], focuses on describing the process by restrictions (so-called *constraints*) over the behaviour, which must be satisfied throughout the whole process execution. Thus, they do not model the control flow explicitly. This paradigm enables a larger degree of flexibility, which is why declarative modelling languages can be considered as well suited for knowledge-intensive processes with many decision points. Declarative process modelling languages provide a repertoire of templates, i.e. a set of commonly used constraint types for modelling knowledge-intensive processes. Various declarative process modelling languages support different process perspectives. There are single-perspective languages (e.g. traditional Declare) as well as languages that support the resource and data perspectives (e.g. MP-Declare). Hence, in multi-perspective modelling languages, constraints are capable of expressing connections between different process perspectives [13,30]. A well-known drawback of declarative modelling languages is that corresponding process models can be hard to read and understand [16]. In the following, several reasons are discussed.

Let us consider the single-perspective process model *hiddenCoExistence* shown in Fig. 1 modelled with a graphical Declare notation. This process model consists of five activities namely a, b, c, d and e. The activities d and e as well as a and b are connected via a *response* constraint which expresses that if activity d or a is executed, activity e or b must be executed eventually afterwards. The *notCoExistence* template between e and b ensures that only one of the two activities is executed. The *precedence* constraint expresses, that c can only be executed if b was executed before.

Having a closer look at the process model reveals implicit dependencies between one or more constraints. These dependencies are called *hidden dependencies* [37]. If activity d is executed, for example, neither b nor c can be executed afterwards. Therefore, the model comprises hidden dependencies between d and b as well as between d and c. In the same manner there is a hidden dependency between e and a. In Sect. 2, we show based on additional examples that hidden dependencies are not an issue with control-flow constraints in particular but also with the declarative modelling paradigm in general. Moreover, in MP-Declare, for instance, this issue is aggravated since it describes additional process perspectives.

A second, more general observation [16] is that it can be hard to interpret declarative process models in terms of concretely permissible process trajectories and the implications of actions in particular states of a running process.

Concluding from the previous explanations, hidden dependencies in particular and a low understandability in general are a consequence from an aspect that is considered to be a core strength of declarative process modelling languages, i.e. its implicit description of permissible process behaviour.

Those issues affect the modelling and maintenance of declarative process models tremendously as well as their execution [17]. When modelling a declarative process there is a high risk for over- or underspecifying the corresponding process model. A process model is called *overspecified* if it forbids valid process trajectories. It is called *underspecified* if it allows process trajectories that are not conformant with reality. Often the reason for a wrong specification of a process model are hidden dependencies or deadlocks, i.e. contradictions between constraints under certain conditions, that were not revealed. The detection of hidden dependencies or deadlocks requires the simulation of process trajectories and their (manual) checking.

When executing a declarative process model, it may happen that executing a specific activity results in a non-executability of an other activity. In our sample process, for example, it becomes impossible to execute activity c if activity d was executed. In other cases, the execution of an activity leads to an extended execution (i.e. it takes more time) or spends more resources. In the worst case, a necessary activity cannot be executed and thus the process instance cannot be completed. Hence, it would be beneficial to know all consequences of a decision to give recommendations to the process participants. For example, the shortest path to finish the process instance can be recommended or the abort of the execution, because the desired outcome of the process is not reachable anymore. However, it is difficult for the process participants to estimate such consequences, even if they know the underlying process model, because of the above mentioned drawbacks of declarative process modelling languages. Based on the problem statement above a research question arises which is the central basis for the rest of the paper: *Given an arbitrary state of a (prospectively) running process, how can we uncover (hidden) dependencies the user might not be aware of?*

The two application scenarios (execution and modelling) mainly differ essentially in terms of time requirements. Using prediction during an execution of a process model it is essential that a prediction is produced almost instantaneously, as the predictions are usually needed in real time. In contrast, a prediction during the modelling phase can be considered to be not time critical.

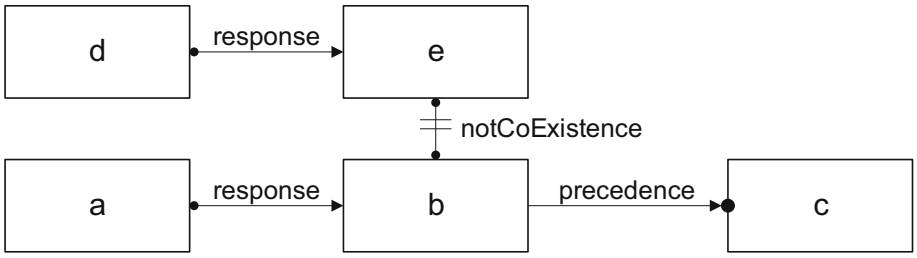

Fig. 1 *hiddenCoExistence* example process in a graphical Declare notation

This paper is a continuation of previous work [25] that takes a first step to a multi-perspective look-ahead functionality that answers the research question above. The look-ahead determines all possible process trajectories up to a desired length based on an uncompleted process execution. While execution engines for single-perspective imperative process models already implement such functionality, execution approaches, for multi-perspective declarative process models that involve constraints on data and resources are less mature and do not consider preventive strategies [3].

Altogether, the present paper contributes (1) a detailed overview about the main challenges of a look-ahead functionality for declarative processes; (2) a concept of a look-ahead functionality that is independent of a concrete process modelling language; (3) a new implementation that outperforms the previous one given in [25]; (4) an experimental evaluation of the implementation on declarative process models; and (5) based on the evaluation, a discussion of strengths and limitations of the look-ahead functionality.

The remainder of the paper is structured as follows. We first recall basic terminology and introduce a running example in Sect. 2. In Sect. 3, we describe the idea of a look-ahead, requirements, challenges and necessary features of a look-ahead functionality in detail. In Sect. 4, we present the concept of a modelling language-independent look-ahead functionality and provide a description of implementation details and all necessary preliminaries. Section 6 gives an overview of related work and delimits our work from existing work. We evaluate our concept in Sect. 5 by means of three declarative process models and inform of existing limitations. Finally, Sect. 7, draws conclusions from the work and gives an outlook for future work.

2 Terminology and running example

In this section, we recall basic terminology and introduce a running example. Process perspectives are introduced to provide a common basis for the contents of both process models and process traces. Traces describe a particular execution state of a process which, traditionally is a completed process trajectory. However, though our look-ahead approach operates on incomplete process trajectories, it is possible to build it upon the same notion.

2.1 Process perspectives

Independent from the modelling paradigm, processes can be seen from different perspectives [29]. Relevant perspectives, for example, include function, behaviour, data, organization and operation [29]. However, it is also feasible to define and consider numerous further perspectives, like a time perspective or a cost-oriented perspective. Considering a process from several perspectives is necessary in order to be able to see a process as a whole and not just to light individual aspects. Otherwise, important factors influencing the process might be neglected. Typically the different perspectives are intertwined, in that sense that they affect each other. We call such coherences *cross-perspective*. For example, often the behaviour depends on value ranges of parameters or organizational characteristics. In the following, we describe the perspectives we will require later more in detail.

The *behaviour perspective* (also called *control flow perspective*) describes activities, their execution ordering (e.g. task sequences or parallelism) or constraints for their execution [29]. Expressing the control flow perspective strongly depends on the underlying modelling paradigm, respectively, the used process modelling language. The *organizational perspective*, in turn, manages the involvement of human resources in processes, e.g. it provides an anchor to the process in the form of human roles responsible for executing activities. Usually, there is a strong relation to an organizational model, that captures process workers, roles, or organizational units as well as their relationships [29]. The *operational perspective* describes technologies and items, e.g. tools or software applications, that are used by performing an activity. The *data-oriented perspective* (also called *information perspective*) deals with a set of data objects as well as the data flow between the activities [29]. Generally, three types of data objects are distinguished: application data, process-relevant data and process control data. Application data is application specific and is usually managed by the process participants or services that are involved in the execution of the process. Typical application data, for instance, are business documents like formular data or contracts. Process-

relevant data encompass data that are used to decide which execution paths shall next be taken (e.g. variables that indicate the state of a process execution) [29]. Hence, such data often reflect pre- and post-conditions of activity execution. The process control data covers information about the current state of a process as well as its execution history (e.g. the time when an activity was executed) [29]. In this paper, most of the time we exclusively deal with application and process-relevant data.

2.2 Events, traces and logs

The approach proposed in the present paper is based on concepts we are going to introduce in this section. We recall the standard definitions of events, traces and process logs as defined in [42]: an event is an occurrence of an activity (a step in a business process) in a particular process instance, a trace is a temporal sequence of events that are related to the same instance of the process, and an event log is a multiset of (completed) traces. Events are characterized by various properties (i.e. information about the execution of an activity), such as the timestamp, the name of the activity, the process participants or systems involved in executing the activity, the used tools or further data payload in form of variables with particular values. A trace is the record of a process execution. Let $\mathcal{E}$ be the universe of all events, i.e. the set of all possible event identifiers. We can assign a value of a property to an event. Though, the proposed approach is independent from any particular set of properties, for the present paper, we assume that these properties are the timestamp ($\mathcal{C}$), the activity name of an event ($\mathcal{A}$), the involved persons ($\mathcal{I}$), the used tools ($\mathcal{T}$) and data objects ($\mathcal{D}$). For a process instance p we denote all process entities that can occur as follows: $\mathcal{C}$ for the domain of timestamps, A_p for the activity names, I_p for persons, T_p for tools, and D_p for further data payload in form of variables. Hence, we can define the following mappings that assign a value of a particular property to an event:

$$\pi_{\mathcal{C}} : \mathcal{E} \to \mathcal{C},$$
$$\pi_{\mathcal{A}} : \mathcal{E} \to A_p,$$
$$\pi_{\mathcal{T}} : \mathcal{E} \to \mathcal{P}(T_p),$$
$$\pi_{\mathcal{I}} : \mathcal{E} \to \mathcal{P}(I_p),$$
$$\pi_{\mathcal{D}} : \mathcal{E} \to \mathcal{P}(D_p).$$

While exactly one activity or timestamp belongs to each event, several tools or persons can be assigned to an event (e.g. a paper is written by two authors with two different word processing tools). Hence, we mapped the event to the power set of the allowed resources. Note, that the power set also contains the empty set, so we can express that no person or no tool is involved in the execution of the activity, by mapping the property to the empty set.

We can now define, trace and event log more formally:

Definition 1 Let $\mathcal{E}$ be the universe of all events. A *trace* is a finite, non-empty sequence $\sigma = \langle e_1, \ldots, e_{|\sigma|} \rangle$ of events of the same process instance such that for $1 \leq i < j \leq |\sigma| : e_i, e_j \in \mathcal{E}; \sigma(i) \neq \sigma(j) \wedge \pi_{\mathcal{C}}(e_i) \leq \pi_{\mathcal{C}}(e_j)$, where $|\sigma|$ denotes the length of σ. We use the notation $\sigma(i)$ to refer to the ith element in σ. An *event log* is a multiset $\left[\sigma_1^{w_1}, \ldots, \sigma_n^{w_n}\right]$ of traces with $w_i \in \mathbb{N}_+$.

Note, that it is important to differ clearly between event and activity. The definition does not disallow the repetition of an activity, it only states that the corresponding event has to be unique within the entire log. Hence, two events can encapsulate the same activity, but then have to differ in other properties like the timestamp, the organizational resource or the affiliation to the process instance. Furthermore, the definition states that all events of a trace refer to the same instance of the process. It also follows from the definition, that the time within a trace is increasing. In case of two events with identical timestamps, the order between these events is chosen arbitrarily.

We say a trace is completed if working on the corresponding process instance was stopped, that means no additional events related to the process instance will occur in the future. In case of imperative process models, this happens by achieving an end event. In case of declarative modelling languages, like Declare, the user has to stop working on the process instance [28]. The user can stop working on a process instance if and only if no constraint of the underlying process model is violated [28].

Definition 2 A trace is called *valid* if the corresponding process instance does not violate a constraint of the process model.

Definition 3 A *completed trace* is a valid trace that was successfully closed, i.e. the user stopped working on the corresponding process instance.

2.3 Running example

In the sequel, we will refer extensively to the following running example.

Example 1 The *publishPaper* process model describes the process of writing and submitting a scientific paper on an abstract level. For a better understanding and since our approach does not depend on a specific modelling language, we initially do not use a concrete process modelling language but a mixture of different languages. We describe the control flow of the process with a graphical Declare notation (cf. Fig. 2). The constraints of the Declare model mean that:

1. an abstract has to be written before its submission,

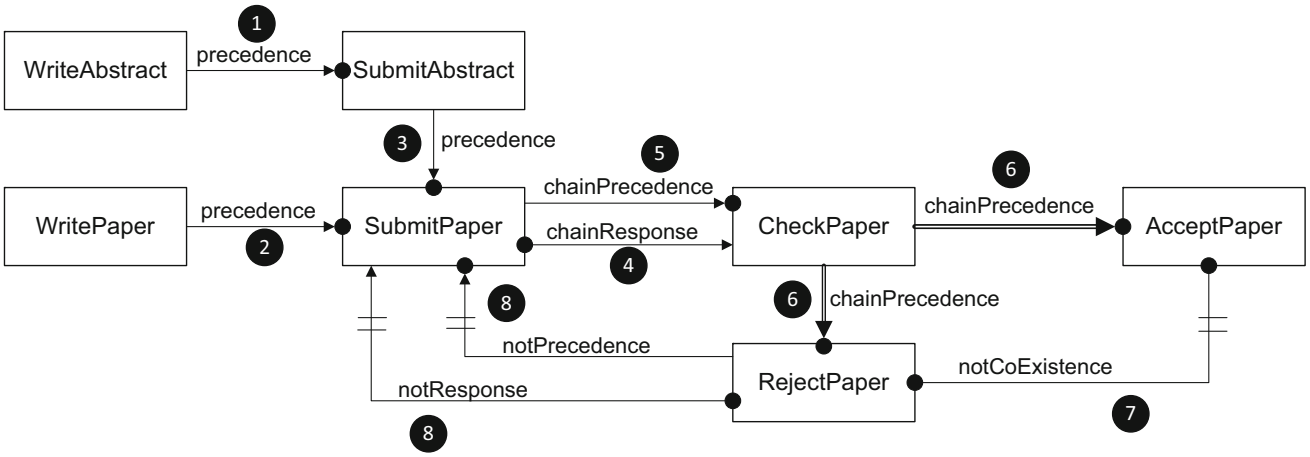

Fig. 2 *publishPaper* example process in a graphical Declare notation

2. a paper has to be written before its submission,
3. the abstract has to be submitted before the paper,
4. a paper must be checked instantly after its submission,
5. before checking a paper, it must have been submitted sometimes before,
6. a paper cannot be accepted or rejected without having been checked before,
7. a paper cannot be accepted and rejected at the same time,
8. a rejected paper should not be submitted again.

The cross-perspective constraints are described in a textual manner in the following:

- an abstract should not exceed the limit of eight lines,
- if the abstract exceeds the limit of eight lines, the submission is rejected instantly,
- a paper should not exceed the limit of ten pages,
- if a paper exceeds the limit of ten pages, it will be instantaneously rejected,
- a paper should not be checked by one of the authors,
- the authors of abstract and paper must be the same,
- a paper, respectively, an abstract should be written from at least one author,
- the authors should use at least one word processing tool.

The process model is multi-perspective and considers with exception of the time perspective, the most common process perspectives.

3 Look-ahead fundamentals

In this section, we discuss the idea of the look-ahead functionality in detail and explain the scope of our approach. Section 3.1 describes the idea of a look-ahead functionality and in Sect. 3.2 we identify requirements and challenges for realizing such a look-ahead functionality referring to our previous work.

3.1 General idea

The *look-ahead* attempts to predict the effects of executing an activity towards possible consequences within an a-priori defined time window. The prediction is based on three artefacts: *(i)* the activity that should be executed next, *(ii)* the underlying process model and *(iii)* the previous trace of the current execution.

Apart from the intended next step activity, we need the process model for the prediction to judge whether an execution is valid. Since the history of a process instance affects the future process trajectory it is essential to consider also the previous trace. Most of the time the previous trace is uncompleted, which means that the process instance is not closed yet (otherwise a look-ahead would be needless).

It is necessary to define a time window, because otherwise the search space for consequences is possibly unlimited. For a unified description, we call this time window *look-ahead window* and its size *look-ahead length*. The look-ahead length indicates the number of events the look-ahead looks in the future. All those terms are visualized in Fig. 3.

The predicted consequences are a broad term and encompass different aspects. These ranges from valid process trajectories including all hereby usable resources like tools or persons to observations like that the execution takes more time, spends more resources or cannot be successfully finished. Furthermore, consequences consider not only things that can happen but also things that become impossible, for example, activities, which cannot be executed again or process trajectories that are not viable anymore. Common consequences would be, for example:

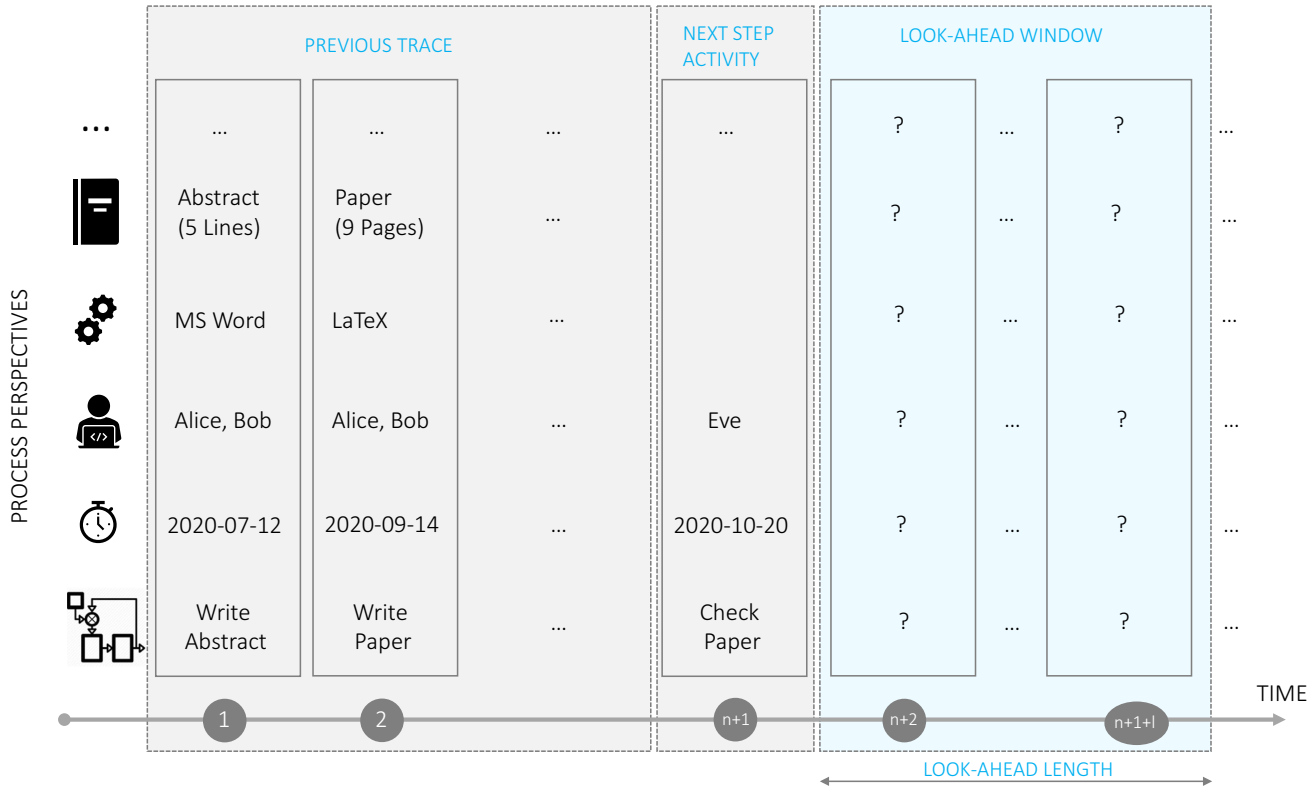

Fig. 3 Illustration of the look-ahead concept

- Which activities can be executed within the look-ahead length?
- Is it possible to complete the process instance within the look-ahead length?
- Is it possible to reach the desired process outcome?
- Which persons and further resources are possibly involved in the execution within the look-ahead window?
- Are there activities or resources that cannot be involved in the future execution of the process instance?
- Is there a deadlock within the look-ahead window?
- Do I have to execute a specific activity next?

We illustrate the idea of a look-ahead by the running example. Assume that the researchers Alice and Bob start writing an abstract by using the word processing software Word. Applying look-ahead of length 1 shows that as a next step Alice and Bob can either submit the abstract or start writing the paper. If we increase the look-ahead length up to 4, we recognize the earliest time for checking the paper and that Alice and Bob are not allowed do this activity by their own. It also follows that it is impossible to achieve the intended process outcome, i.e. the acceptance of the paper, within the look-ahead window. The effect of the look-ahead length becomes more clearly by the following scenario: assume that Alice and Bob have exceeded the page limit of the paper. The look-ahead of length 4 shows that the process instance can go on and reveals no problems. However, if we look 5 steps into the future we recognize that it is unavoidable that the paper is rejected.

3.2 Challenges and requirements

In this section, we first present some challenges which the development of a look-ahead functionality for multi-perspective declarative process models bears. Afterwards, we derive requirements from these challenges that the look-ahead functionality should meet. We use these requirements later to proof our claims.

Challenge 1: Large number of possible process trajectories The provided flexibility of declarative (multi-perspective) process models goes hand in hand with a tremendous growth of possible process trajectories. Their number is not only affected by the control-flow perspective but also by further process perspectives. In the publishPaper example (cf. Sect. 2.3), for instance, the possibility to use different word processing tools for writing increases the number of valid process trajectories. Let us consider the following traces:

$\sigma_1 = \langle$WriteAbstract(Alice; Word), WritePaper(Alice; Word)$\rangle$,
$\sigma_2 = \langle$WriteAbstract(Alice; Word), WritePaper(Alice; Latex)$\rangle$,
$\sigma_3 = \langle$WriteAbstract(Alice; Word), WritePaper(Alice; Word, Latex)$\rangle$.

We observe that all traces are identical w.r.t. the control-flow perspective, but differ on the operational perspective. The situation is aggravated if the data-oriented perspective is taken into account. Since the paper must not exceed the limit of 10 pages, for each of the three traces there exist 10 valid process trajectories. Assume we can additionally choose between three authors and two word processing tools. Hence, we would have $7 \cdot 3 \cdot 10 = 210$ possibilities for a correct execution of the activity WritePaper. In our calculation, the first factor describes all non-empty subsets of possible authors, in the same way the second factor is the number of combinations of word processing tools, and the last factor indicates the number of values for the number of pages that are still within the given page limit. In other words: starting with writing the abstract like in the traces above leads to at least 30 valid execution options (note that the author is fixed, since paper and abstract must have the same authors). In this scenario, it would also be possible to submit the abstract. For this activity, there exists only one possibility. Since there are processes where variables can attain an arbitrary integer or real value, this could lead to an infinite number of process trajectories. Hence, we have to make appropriate assumptions to limit the approach to a finite number of process trajectories without a significant loss of quality.

Challenge 2: Checking validity of process trajectories The particularities of modelling languages affect the validation of process trajectories. The manner of checking whether a trace violates a constraint depends on the constraint itself. For example, some constraints must be checked forwards, others backwards and some of them by analysing the complete process trajectory [36]. Assume there would be a constraint that demands that the paper can only be checked once. In this case, the fulfilment can only be checked by considering the whole trace. In case of the precedence constraint, that an abstract has to be written before its submission it would be sufficient to consider the events in the trace before the event that contains the SubmitAbstract activity.

Challenge 3: Intertwined process perspectives The structure of the constraints together with the multi-perspectivity make it impossible for validation to consider the process perspectives isolated or to neglect some of them. Neglecting one or more process perspectives would reduce the number of constraints, which would be beneficial for validation (since less constraints must be checked). In our example, the control flow is affected by the data-oriented perspective, since exceeding the maximum limit of pages of the paper leads to a rejection of the submission. Focussing only on the control-flow perspective reveals no violations. The violation only affects the data-oriented perspective. Another example would be the following (uncompleted) trace:

$$\sigma_4 = \langle \text{WriteAbstract(Alice; Word)}, \text{WritePaper(Bob; Word)} \rangle.$$

Regarding the control-flow perspective, the trace is valid. However, there is a violation on the organizational perspective, since the authors of both activities must be the same. Hence, it is not possible to omit process perspectives in general. A process perspective can only be neglected for particular traces and then only if a violation of a constraint on other process perspectives has already been identified.

Challenge 4: Multiple changes of validity Especially in case of a long look-ahead length, it would be beneficial for the performance if an uncompleted trace which is currently invalid could be labelled as invalid for all time. Then this trace could be excluded for further consideration. However, traces can change their validity several times. For example, we consider the evolution of a trace σ_5 at different times (t_1, t_2, and t_3):

$$\sigma_5^{t_1} = \langle \text{WriteAbstract, WritePaper, SubmitAbstract} \rangle,$$
$$\sigma_5^{t_2} = \langle \text{WriteAbstract, WritePaper, SubmitAbstract, SubmitPaper} \rangle,$$
$$\sigma_5^{t_3} = \langle \text{WriteAbstract, WritePaper, SubmitAbstract,}$$
$$\text{SubmitPaper, CheckPaper} \rangle.$$

We consider that $\sigma_5^{t_1}$ is valid, but after submitting the paper ($\sigma_5^{t_2}$) the trace becomes temporary invalid, because after submitting paper and abstract a check of the paper is mandatory. Executing the activity CheckPaper ($\sigma_5^{t_3}$) changes the validity of the trace into valid again. Hence, one of the key challenges is to recognize whether an invalid uncompleted trace stays invalid or not.

Challenge 5: Length of the look-ahead window It has already been mentioned in Sect. 3.1 that the look-ahead length affects accuracy and reliability of the prediction, since a greater look-ahead length gives more insights and consequences. However, a large look-ahead length is potentially accompanied by a strong growth of possible process trajectories. A crucial task therefore is to determine the optimal look-ahead length for the given uncompleted trace. Already determining an upper bound is no trivial task, since the execution of a process instance can be extended or shortened through the execution of a particular activity.

These challenges lead to four main requirements the look-ahead functionality should meet:

Requirement 1: Completeness of simulation Within the mandatory limitation described in Challenge 1 the look-ahead should find reliably all valid process trajectories (so-called completeness of simulation). Otherwise it is not sure that all relevant consequences can be derived. In some cases, it seems to be sufficient to search only for a subset of good enough process trajectories (because some traces are very similar or of low interest). However, the decision what can be considered as good enough depends on the particular situation and the particularities of the process. Since our approach should work independently of such particularities, we require all process trajectories. Nevertheless, there should

be some appropriate filter methods to extract the relevant process trajectories by individual criteria.

Requirement 2: Systematic and efficient search The large number of possible process trajectories requires a systematic and efficient search within the look-ahead window. This includes among other things avoiding redundancy, in that sense that execution paths should not be checked multiple times.

Requirement 3: Language Independence There are several different modelling languages for (multi-perspective) declarative processes that suffer from the understandability problems described in the introduction. Hence, our look-ahead approach is not only relevant for selected modelling languages and must be independent of a particular language as far as possible. This bears its own challenges, because different languages possess different expressiveness (we will discuss this issue in Sect. 4.1.2) and their own particularities. For example, in MP-Declare and Declare the repository of templates is incomplete in that sense, that both languages provide the possibility to define custom templates by using the underlying logic (Metric First-Order Temporal Logic [8]). Furthermore, the support of different process perspectives between the languages differ. Hence, the look-ahead should support as much as possible the particularities of different process modelling languages and should be extendable to further languages.

Requirement 4: Output format for easy post-processing In general the output of the look-ahead should be in a format that allows an easy post-processing with existing tools and frameworks to derive different consequences. As seen in the previous section, those consequences are manifold, so the output format should be supported by numerous frameworks and tools and must enable an easy way for implementing custom types of analysis.

4 Concepts of simulation-based look-ahead

We use a simulation-based approach for implementing the look-ahead functionality described in Sect. 3. The term simulation-based means that the proposed approach generates all possible continuations of a process instance (future process trajectories) within the look-ahead window, in case of executing the intended activity next. The generated process trajectories are eventually validated whether they conform to the underlying process model. Hence, the output of the look-ahead is a set of traces (these can be either completed or uncompleted, since the trace length is restricted by the look-ahead length). This output is sufficient for our purposes, since the traces contain all necessary information to derive possible consequences as exemplarily mentioned in Sect. 3.1. We based our simulation technique on a transformation of declarative process models to a logic language called Alloy [24]. The Alloy language is a declarative modelling language for describing software structures, taking into account desired or required restrictions [24]. Hence, this objective is very similar to the idea of declarative process modelling languages. The usage of a logic framework is in some way natural, because modelling languages like Declare or MP-Declare are based on logics. In a strict sense, Alloy is not just a logic language rather than a framework that consists of a logic, a language and an analysis tool (*Alloy Analyzer*) [24]. We use the well-known *eXtensible Event Stream (XES)*[2] standard format to store the generated traces, because this format enables a flexible analysis, since nearly every process mining tool is built upon this format.

In addition to the activity that should be executed next, the underlying process model, and the previous trace of the current execution, the look-ahead requires as input all activities and resources (e.g. persons, tools or variables) that can be theoretically involved in the execution. Otherwise, it would be not clear which concrete instances of the resources (also called process entities) can be used within the process, respectively, the process instance. Note, that the process model does not contain this information explicitly. For example, in the publishPaper process model we have constraints about authors and reviewers, but no concrete instances of authors or reviewers. For simulation, we have to explicitly specify these instances, e.g. that all possible authors are Alice, Bob and Eve and all possible word processing tools are Word, Latex and LibreOffice. Our approach is visualized in Fig. 4. In the following, we describe the single steps of the approach including all necessary preliminaries in detail.

4.1 Transformation to Alloy

For the simulation, we have to transform the inputs of our approach to an Alloy specification. Since we utilize Alloy as the language to specify the simulation task, i.e. the language that describes the task for the look-ahead, it is necessary to be able to represent a process execution trace in Alloy. Therefore, we describe an Alloy-based meta-model (so-called *Process Event Chain Meta-Model*). This meta-model was proposed in [2] and extended in [3] to support the data-oriented perspective. We adapt this meta-model from a previous version [3] to support also the operational perspective. In summary, it covers up the following process perspectives: *(i)* functional and control-flow perspective, *(ii)* organizational perspective, *(iii)* operational perspective and *(iv)* data-oriented perspective. In the subsequent section we describe this meta-model in detail, since it is fundamental to the understanding of the transformation to Alloy. Afterwards, we describe the transformation of process model, process entities, previous trace and next step activity to Alloy.

[2] http://www.xes-standard.org/openxes/start.

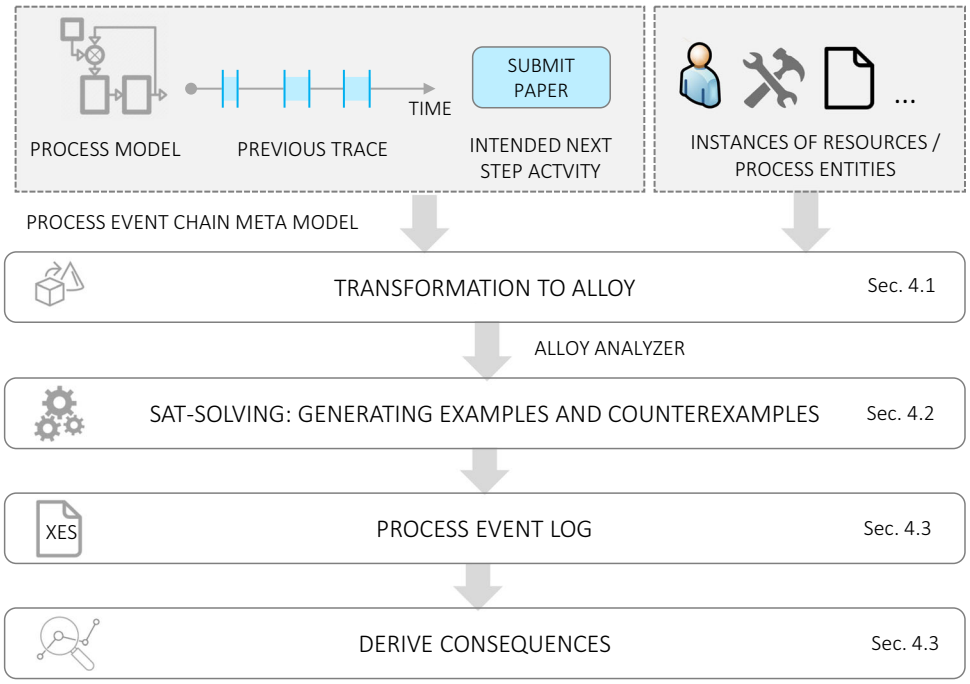

Fig. 4 Concept of the simulation-based look-ahead

4.1.1 Process event chain meta-model

The structure of the meta-model is shown in Fig. 5. In order to represent an event in Alloy, the abstract signature *PEvent* is introduced. This abstract signature forms the backbone of the chain structure of the process execution traces. A signature in Alloy describes a static structure and can be compared with a class in object-oriented programming languages. Each *PEvent* just contains its unique position (*pos*) within the trace. It would be more intuitive to implement this trace as a singly or doubly linked list. However, a performance test showed that such an implementation is accompanied by a significantly worse runtime [2]. It is worth to notice that in Alloy the keyword abstract has a different meaning than in object-oriented programming languages. In Alloy the keyword abstract does not automatically prevent an instantiation of the signature. An instantiation is only prevented if there is another signature that extends the abstract signature [24]. Hence, all events in the trace are instances of *TaskEvents* (the last extension of *PEvent*).

The signature *TaskEvent* inherits from *PEvent* and encapsulates all resources that are carried by an event (e.g. the name of the activity or involved persons). This encompasses at least one instance of *AssociatedElement*. The activity referring to the event is modelled as an instance of the signature *Task*. The signature *AssociatedElement* serves as an interface that enables the extension of the meta-model by additional process elements to support further process perspectives.

To enable the modelling of organizational structures, we integrate the organizational meta-model that was first proposed in [9] and transformed to Alloy in [2] in our meta-model. It adds the ability to assign persons (*Identity*) to events and to build hierarchical organizational structures. The operational perspective is realized in the same way via the extension *Tool* of the signature *Element*. The signature *Relation* allows to formulate structural relations between all subtypes of *Element*. Therefore, *Relation* contains three fields: subject, predicate and object. An example of such a relation would be: Alice (subject) hasRole (predicate) author (object). Since the meta-model allows that every subtype of *Element* can act both as subject and object, it is up to the modeller to formulate meaningful relations.

The meta-model provides the functionality for representing the data-oriented perspective through the interaction of the signatures *DataObject*, *WriteAccess*, *VariableObject* and *Value*. A *DataObject* can represent, for instance, a document or a formular, that can be named by extending it or variables (*VariableObject*) that form further payload, like the maximum page limit in our running example. A *WriteAccess* assigns a concrete value (an instance of the signature *Value*) to a data object. Therefore, a *WriteAccess* contains a field which refers to a data object and a field for its assigned value. The value can be either an integer or a character sequence (so-called String). Other data types, such as floating-point numbers are not supported by Alloy. However, we can simulate a floating-point number with two integers, one representing the significand and one the exponent (power of ten). The data-oriented perspective in the meta-model would be expressive enough to encode the time perspective of a process by representing the time as one or more integer values. This would be the same concept as programming languages use for representing time (cf. unix timestamps).

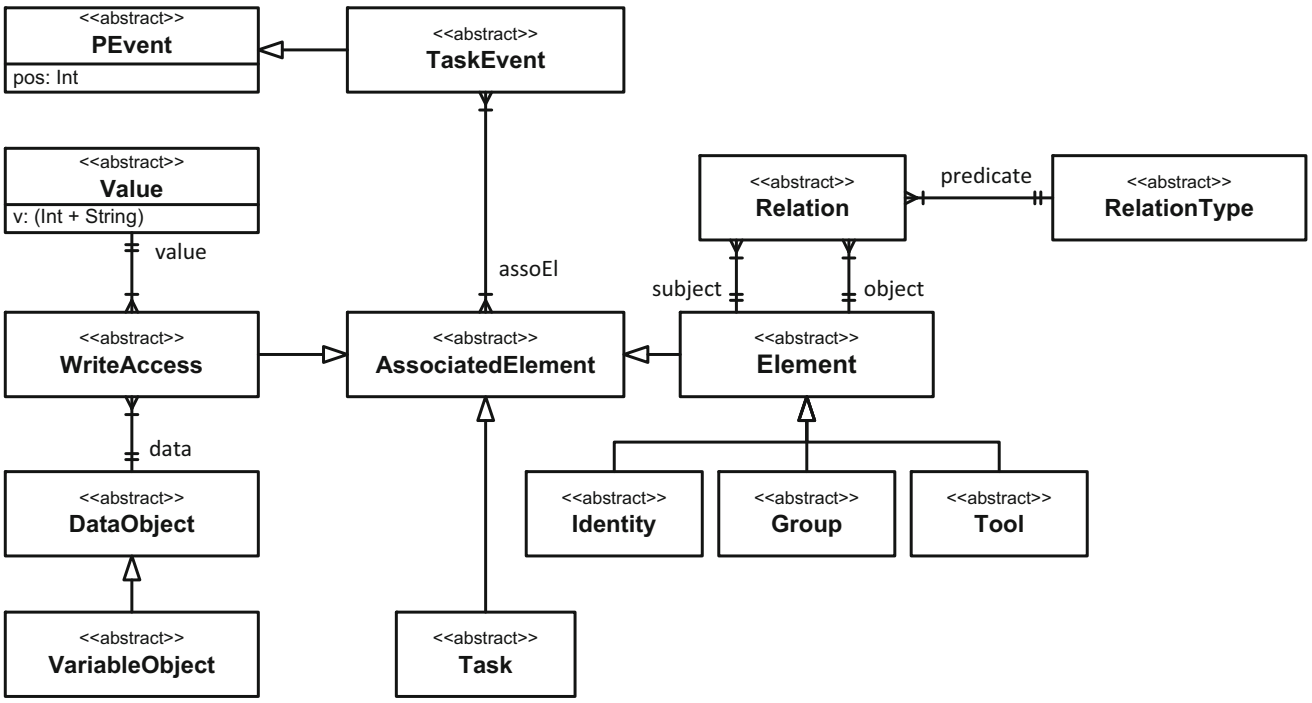

Fig. 5 Process Event Chain Meta-Model

Note, that in this approach it makes no difference whether we consider time as global, local, relative or absolute times.

The meta-model was implemented in Alloy. In order to keep the description of preliminaries concise, we omit presenting the source code and refer to [3]. The implementation contains some additional non-structural constraints to express important restrictions that cannot be covered up by the static structure of the meta-model. These are mentioned in the following:

1. A *TaskEvent* is associated with exactly one activity (*Task*).
2. The *PEvents* are consecutively numbered and start at the lowest available number, to ensure that the trace is continuous and gap free.
3. All *AssociatedElements* can only occur together with an event.
4. A *Group* can only be part of a process instance if their contained organizational units are also part of the process instance.
5. An activity cannot be executed by a group, only by the members (identities) of the group.
6. Each data object can occur at most one time in an event (together with the concept of the *WriteAccesses* this prevents that the value of a data object is changed multiple times within an event).

4.1.2 Transformation of multi-perspective process models to Alloy

The transformation of a declarative multi-perspective process model to Alloy is based on the initial modelling language. This step was already done for MP-Declare [3] and DPIL [2]. We would like to briefly recap the essential steps of transforming an MP-Declare model and a DPIL model to Alloy. Afterwards, we discuss under which preliminaries this approach can be extended to cover further modelling languages like DCR graphs. The transformation of an MP-Declare or DPIL process model involves two major steps: *(i)* creating signatures for the process entities and *(ii)* transforming the constraints (MP-Declare), respectively, rules (DPIL) of the process model to Alloy. In order to be able to follow the contribution we explain those parts of the Alloy notation that are necessary for understanding.

The process entities are transformed to Alloy code as described in Algorithm 1. Therefore, each process entity is transformed into a unique and eponymous signature extending the particular entity type (e.g. Task, Tool, Identity or VariableObject).

In the literature, MP-Declare is considered as a repository of constraint templates. However, this repository is not complete and can be extended by defining custom templates with the underlying Metric First-Order Temporal Logic. In [3] a one-to-one mapping from MP-Declare templates to Alloy predicates is presented. In Alloy a predicate describes

Algorithm 1: Transformation of process entities to Alloy code

Input: Activities A_p, tools T_p, persons I_p, data objects D_p
Output: Alloy code that initializes the process entities

```
for a ∈ A_p do
    print one sig a extends Task {}
end
for t ∈ T_p do
    print one sig t extends Tool {}
end
for i ∈ I_p do
    print one sig i extends Identity {}
end
for d ∈ D_p do
    print one sig d extends VariableObject {}
end
```

```
pred response[t1, t2: Task, act, tar:
    TaskEvent, cor: TaskEvent -> TaskEvent]
    {
    ∀ hte: any[t1] | #(inAfter[hte, t2] ∩
        act ∩ cor.hte) > 0
}
```

Listing 1 Alloy formulae of the MP-Declare response template

a function with a Boolean return value [24]. This mapping encompasses the common MP-Declare templates in the literature and must be extended for custom templates. The idea behind this mapping is based on two concepts: *(i)* temporal relationships between events are mapped to positional relationships in a trace and *(ii)* additional attribute conditions are mapped to a constraint that selects a subset of all existing events based on their attributes. According to the structure of the templates, the predicates have as parameter one or two activity events and additional restrictions like activation, target and correlation conditions. A repository of the common MP-Declare templates and their corresponding Alloy formulae can be found in [3]. For better understanding of this mapping Listing 1 shows exemplarily the MP-Declare response template as corresponding Alloy formulae. The concept for transforming DPIL rules to Alloy is identical. A table of DPIL rule templates and corresponding Alloy formulae can be found in [2]. Using these transformation rules, it is straightforward to transform MP-Declare rules to Alloy, since each rule is an instance of an MP-Declare template. These instances are listed in an Alloy fact block. In Alloy *facts* are blocks for formulating invariants, i.e. they contain constraints that are always assumed to be true [24]. It does not matter whether one or more fact blocks are used. Hence, we can form a conjunction of constraints to represent the whole process model. In order to automatically translate MP-Declare models and DPIL models to Alloy a model-to-text transformation was implemented.

The underlying logic for Alloy is a threefold calculus: *first-order predicate logic*, *relational logic* and *navigation expression logic* [24]. Hence, the expressiveness of Alloy is limited to first-order logic. Consequently, it is only possible to map modelling languages to Alloy that possess a expressiveness that is less or equals to first-order logic (FOL). Otherwise, not all features of a language can be mapped to Alloy. In case of Declare, which relies on Linear temporal logic (LTL) [28], it was shown in [18] that LTL and first-order logic are equally expressive over discrete-time models. However, the expressiveness of multi-perspective declarative modelling languages is an open question in research and would exceed the scope of this paper. Because of the successful use for MP-Declare and DPIL, an Alloy-based approach seems promising for other modelling languages as well.

4.1.3 Transformation of the previous trace to Alloy

The Process Event Chain Meta-Model allows the representation of a trace as an instance of the meta-model. We represent the next step activity as an event e_{next} that additionally contains all process entities that would be involved by performing this activity. Since the output of the look-ahead is a set of continuations of the previous trace extended by e_{next}, we can simply append this event to the previous trace. Hence, we have no longer to consider a previous trace and an intended next step activity separated from each other (which is why we only use the term trace in the following). We use a fact block to represent this trace in Alloy, which means, the Alloy Analyzer interprets this trace in the same way as a constraint of the process model.[3] Hence, all generated traces have to fulfil this constraint, in that sense that the trace must be a prefix of the simulated traces.

The position of an event in the process event chain is identified via an index (field *pos* in the meta-model). In Alloy it is necessary to define a-priori the number of available integer values. This is done via the bitwidth parameter B. The Alloy Analyzer then generates integer values in the range of

$$\left[-2^{B-1}, 2^{B-1}-1\right].$$

It is an Alloy specific peculiarity that the indices are arranged symmetrically around zero [24]. Hence, the first position of the trace has the index -2^{B-1}. For example, a bitwidth of 6 allows a maximum trace length of 64 and the indices range from -32 to 31.

This limitation of integers also affects the representation of data objects, since it requires finite domains and the variables can attain only integer values within in the above defined range. Hence, we have to choose the bitwidth large enough

[3] Remember that the Alloy Analyzer searches for examples that fulfil a given Alloy model and the given Alloy model here describes traces, which means that the Analyzer will generate exemplary traces.

Algorithm 2: Transformation of previous trace and next step activity to Alloy

Input: Previous trace σ_{prev}, next event e_{next}, bitwidth B, activities A_p, tools T_p, persons I_p, data objects D_p
Output: Alloy code that describes the previous trace and the next step activity

```
1  σ ← σ_prev.append(e_next)
2  start ← −2^(B−1)
3  k ← 1
4  while k < |σ| do
5      print fact {
6      print #(atPos[π_A(σ(k), start + k)]) > 0
7      for t ∈ T_p do
8          if t ∈ π_T(σ(k)) then
9              print #(atPos[t, start + k]) > 0
10         else
11             print #(atPos[t, start + k]) = 0
12         end
13     end
14     for i ∈ I_p do
15         if i ∈ π_I(σ(k)) then
16             print #(atPos[i, start + k]) > 0
17         else
18             print #(atPos[i, start + k]) = 0
19         end
20     end
21     print #(nWaAtPos[start + k]) = |π_D(σ(k))|
22     if π_D(σ(k)) ≠ ∅ then
23         for v ∈ π_D(σ(k)) do
24             if δ(σ(k), v) ≥ 0 then
25                 print #(atPos[wa_v_δ(σ(k), v), start + k]) > 0
26             else
27                 print
                   #(atPos[wa_v_n|δ(σ(k), v)|, start + k]) > 0
28             end
29         end
30     end
31     print }
32     k ← k + 1
33 end
```

to encompass the domains of the data objects. We refer to the domain of a data object v as dom_v. Since Alloy only supports integer values, the domain is a subset of $\mathbb{Z}$ and can be defined via an enumeration (e.g. $dom_{\text{Abstract}} = \{1, 2, \ldots, 8\}$). We define the following mapping to return the attained value of a particular data object within an event:

$$\delta : \mathcal{E} \times D_p \to \mathbb{Z}, (e, v) \mapsto a \in dom_v.$$

Based on the previous definitions, it is now possible to describe the algorithm which transforms a trace to Alloy (cf. Algorithm 2). After appending the next step event to the previous trace (Algorithm 2 line 1), we have to iterate through the trace (Algorithm 2 line 4) and assign the events including their involved process entities to the corresponding position. We use the utility function *atPos(assoElement, pos)* to assign an arbitrary subtype of *AssociatedElement* to an event at a certain position. A function in Alloy is a reusable snippet of code that can be parameterized via zero or more arguments and which returns a set or an integer [24]. The *atPos* function gets two parameters (the AssociatedElement and the position of the event) and returns a set of *TaskEvent*s which satisfy two conditions: the position of the *TaskEvent* must be identical to *pos* and *assoElement* must be associated with the event. The correct position is calculated by shifting the index k of the trace by -2^{B-1}. The assignment to the event on the given position is ensured as we demand that the return value of this function should not be the empty set, i.e. the cardinality (in Alloy denoted by #) of the returned set must be greater zero (e.g. Algorithm 2 line 6).

However, it is not sufficient to assign all process entities to the correct positions, since that does not prevent a later modification of the trace through the Alloy Analyzer. Since we modelled the trace as a fact block the Alloy Analyzer has to fulfil the assignments. Hence, associated elements cannot be removed from an event. However, it is feasible for the Alloy Analyzer to add additional elements to the events of the trace. For example, a further word processing tool or author could be added to the event that contains the activity WriteAbstract. Since the historic execution of a process instance must be immutable we have to explicitly exclude all process entities that are not related to the respective event. This is done in different ways for the types of the process entities. For activities (type Task), it is not necessary to exclude other activities, since the structure of meta-model ensures that each event contains exactly one activity. In case of tools and identities, we prevent assigning further elements to an event utilizing the *atPos* function in the following way: we demand that for all tools and identities that are not associated explicitly with the event at a certain position the *atPos* function should return the empty set, i.e. a set with cardinality zero (Algorithm 2 lines 11 and 18).

s

The assignment and the prevention of undesired assignments of data objects to an event requires a more sophisticated handling than assigning performers, activities and tools to an event (Algorithm 2 lines 21–30). Since a data object and its concrete value are encapsulated within a WriteAccess, the WriteAccess is assigned to an event instead of the data object. However, this encapsulation is concerned with some performance issues. During the simulation of project trajectories the Alloy Analyzer would have to create WriteAccesses on demand. A performance test has shown that the creation of a WriteAccess during simulation runtime takes significant more time than assigning already instanciated WriteAccesses to an event. Hence, we create and initialize a-priori all theoretically possible WriteAccesses, so that during runtime the WriteAccesses simply have to be assigned to events. This procedure is described in Algorithm 3 which is executed before Algorithm 2. The algorithm creates a WriteAccess for

Algorithm 3: Creating and initializing WriteAccesses for data objects

```
Input: Data objects D_p
Output: Alloy code that initializes all possible WriteAccesses
        for data objects
1  for v ∈ D_p do
2      for j ∈ dom_v do
3          if j ≥ 0 then
4              print one sig wa_v_j extends WriteAccess {}
5              print fact {
6              print wa_v_j.data = v
7              print wa_v_j.value = j
8              print }
9          else
10             print one sig wa_v_n|j| extends WriteAccess {}
11             print fact {
12             print wa_v_n|j|.data = v
13             print wa_v_n|j|.value = j
14             print }
15         end
16     end
17 end
```

each combination of data object and value of its domain. The creation involves two steps. First, a signature for the WriteAccess is created (Algorithm 3 line 4 and line 10). The name of the WriteAccess is composed by the prefix *wa*, followed by the name of the data object, and the value. Since negative integers would violate the Alloy convention for signature names, we have to consider whether the value is greater or equal to zero or negative (Algorithm 3 line 3). In case of a negative value, we take the absolute value with the prefix *n* (Algorithm 3 line 10). Second, the WriteAccess is initialized in a fact block, associating the corresponding data object and the particular value (Algorithm 3 lines 5-8 and 11-14). Eventually, the WriteAccesses can be assigned under consideration of their names to a certain position via the *atPos* function (Algorithm 2 lines 24–28). The meta-model ensures that every data object can occur at most once within an event. Hence, we have to prevent assigning further data objects only. This is done by limiting the number of WriteAccesses on the corresponding position in the trace via the function *nWaAtPos(pos)* to the number of assigned data objects (Algorithm 2 line 21).

We demonstrate the transformation of a trace to Alloy by the following example. Assume we have the following previous trace (we indicate the attained value of a data object in squared brackets directly after the name of the data object)

$$\sigma = \langle \text{WriteAbstract(Alice, Bob; Latex; Abstract[2]),} \\ \text{SubmitAbstract(Alice Bob; Abstract[2])} \rangle$$

and a next step activity

$$e_{next} = \text{WritePaper(Alice, Bob; Word; Paper[8]).}$$

Furthermore, the process entities are assumed to be as follows:

$$T_p = \{\text{Latex, Word, LibreOffice}\},$$
$$I_p = \{\text{Alice, Bob, Eve}\},$$
$$D_p = \{\text{Paper, Abstract}\}.$$

As described in the running example, we have the following domains for the data objects:

$$dom_{\text{Paper}} = \{1, 2, \ldots, 10\},$$
$$dom_{\text{Abstract}} = \{1, 2, \ldots, 8\}.$$

Applying Algorithms 1, 3 and 2 in exactly this order with the inputs above and a bitwidth of 6 returns the Alloy code shown in Listing 2. Note, that this Alloy code contains all inputs for our approach.

```
//Process entities (Ap, Tp, Ip, Dp)
one sig WriteAbstract extends Task {}
...
one sig RejectPaper extends Task
one sig Alice extends Identity {}
one sig Bob extends Identity {}
one sig Eve extends Identity {}

one sig LibreOffice extends Tool {}
one sig Word extends Tool {}
one sig Latex extends Tool {}

one sig Paper extends VariableObject {}
one sig Abstract extends VariableObject {}

//WriteAccesses for Paper
one sig wa_Paper1 extends WriteAccess {}
fact {
    wa_Paper1.data = Paper
    wa_Paper1.value= 1
}
...
one sig wa_Paper10 extends WriteAccess {}
fact {
    wa_Paper10.data = Paper
    wa_Paper10.value= 10
}

//WriteAccesses for Abstract
one sig wa_Abstract1 extends WriteAccess {}
fact {
    wa_Abstract1.data = Abstract
    wa_Abstract1.value= 1
}
...
//Process Model
...
//Trace including next event
fact {
    //Previous Trace
    #(atPos[WriteAbstract,-32])>0
    #(atPos[LibreOffice,-32])=0
    #(atPos[Word,-32])=0
    #(atPos[Latex,-32])>0
    #(atPos[Bob,-32])>0
    #(atPos[Eve,-32])=0
    #(atPos[Alice,-32])>0
    #(nWaAtPos[-32])=1
    #(atPos[wa_Abstract2,-32])>0

    #(atPos[SubmitAbstract,-31])>0
    #(atPos[LibreOffice,-31])=0
    #(atPos[Word,-31])=0
    #(atPos[Latex,-31])=0
```

```
        #(atPos[Bob,-31])>0
56      #(atPos[Eve,-31])=0
        #(atPos[Alice,-31])>0
58      #(nWaAtPos[-31])=1
        #(atPos[wa_Abstract2,-31])>0
60
        //Next event
62      #(atPos[WritePaper,-30])>0
        #(atPos[LibreOffice,-30])=0
64      #(atPos[Word,-30])>0
        #(atPos[Latex,-30])=0
66      #(atPos[Bob,-30])>0
        #(atPos[Eve,-30])=0
68      #(atPos[Alice,-30])>0
        #(nWaAtPos[-30])=1}
70      #(atPos[wa_Paper8,-30])>0
    }
```

Listing 2 Example for representing a trace in Alloy

4.2 SAT-solving: generating examples and counterexamples

Through the transformation of the process model and the current process execution state to Alloy, we have transformed a context-aware process simulation problem into a SAT problem. The Alloy Analyzer provides different constraint solver (so-called *SAT solvers*) for solving SAT problems. So far, we have introduced Alloy's SAT solvers as techniques that generate examples for a given Alloy model. However, more precisely it is possible to run them in two opposed modes, i.e. the SAT solvers can be used to generate instances that either satisfy or violate a given Alloy specification. We refer to the former as examples and to the latter as counterexamples. Therefore, we have to configure a command which causes the Alloy engine to search for instances. This can be done according to the following templates depending on whether we are searching for examples or counterexamples.

In case of searching examples, we define an empty predicate (*look*) and configuring a run command: `pred look run look for x TaskEvent, B Int`. If one is interested in counterexamples, we have to define an assert (*look*) and must configure a check command: `assert look hypothesis check look for x TaskEvent, B Int`.

The configuration of the solution space is for both commands identical. The solution space is called *scope*. Through defining the scope we bound the sizes of the sets assigned to type signatures. This limitation of the solution space is necessary to make the SAT problem finite. Otherwise, the SAT problem might be undecidable due to the infinite amount of possible solutions. Since the look-ahead window represents a limited solution space, this limitation does not affect the functionality of the look-ahead. Nevertheless, the scope must be specified correctly for generating the process trajectories. Therefore, we must specify the number of events (x). This parameter is set to the sum of the length of the previous trace including the next step activity and the look-ahead length. In the case that we are searching for counterexamples, we must specify in the *assert* block what exactly we are looking for a counterexample (*hypothesis*). If we want for example answer the question (cf. Sect. 3.1): Do I have to execute activity a next (i.e. at the current position *currentPos*)? It is necessary to encode the question as a hypothesis in the assert block. This hypothesis can be formulized by assuming an answer to the question. In our example, the hypothesis would be: suppose we execute a next (of course it would be also possible to suppose that we do not execute a next). This could be encoded in the following way: `assert look atPos[currentPos, a]`. Note, that we can arbitrary combine constraints in the assert block, to ask for counterexamples for more complex questions that also relates identities, data objects and tools.

Within this defined scope Alloy guarantees distinctness and exhaustiveness while generating solutions. These properties avoid redundant solutions and ensure that all possible solutions are considered. As a consequence of these two properties Alloy also achieves determinism. Determinism means that the found solutions can be replicated according to the predefined settings [2].

In case of counterexamples, one is often interested only in one or a fixed number of process trajectories. Therefore, we add the possibility to generate only a fixed number of process trajectories. However, the quality of the generated solutions is unclear. Hence, it could be necessary to generate a lot of counterexamples until we found a solution that fits to the requirements.

4.3 Transformation to XES and deriving consequences

We export the generated traces of the previous step into the well-known *eXtensible Event Stream (XES)* standard format for further analysis. Therefore, the solutions found by the Alloy Analyzer must be parsed and converted to the XES format. The XES format is well suited for deriving consequences, since nearly every process mining tool is built upon this format. Hence, we can use the rich set of different process mining techniques to derive consequences of various kinds and can adapt the analysis to custom preferences. This flexibility is important, because the focus of interest can be manifold (cf. Sect. 3.1) and each process contains some individual particularities that need to be dealt with. Note, that it makes no difference whether we analyse valid traces or counterexamples in form of invalid traces. This distinction is only important for the interpretation of the results.

When parsing the solutions of the Alloy Analyzer each trace is labelled with a unique ID. In dependence on the encompassed process perspectives of the simulated process model, we add the necessary XES extensions to the event log. Furthermore, we enriched our look-ahead functionality with three of the analyses described in Sect. 3.1:

- *Is it possible to finish the process instance within the look-ahead window?* For answering this question, we check whether no solution is found or a completed trace within the look-ahead window. In case of no solution, it is impossible to finish the process instance within the look-ahead window.
- *Which process entities are possibly involved in the execution within the look-ahead window?* We iterate through the event log of generated traces and extract the process entities used within the look-ahead window. Therefore, it is sufficient to skip the previous trace including the next step activity.
- *Is it possible to reach the desired process outcome?* The outcome is described via a particular activity and additional requirements on other process perspectives. Hence, we have to check whether a trace fulfils these conditions.

We conclude this section with one important remark about the interpretation of the look-ahead results. All results must be interpreted against the background of the limitation of the look-ahead window, since it affects the set of generated traces in many ways. For example, we cannot conclude that every process entity that is not involved within the look-ahead window is automatically forbidden within this scope. The reason for that is that the validity of a trace can—as mentioned in Sect. 3.2—change several times. Since we are only interested in valid process trajectories, we neglect all those trajectories that are invalid within the look-ahead window but can become valid again. Note, that in case of counterexamples it is the other way around and we are interested in invalid process trajectories. Hence, with a larger look-ahead window some further process entities could still be involved.

4.4 Reasons for Alloy and comparison with other logic frameworks

In the following, we give some reasons for using Alloy and compare it with other logic frameworks. Recall that the Alloy logic is a threefold calculus: *first-order predicate logic*, *relational logic* and *navigation expression logic* [24]. Comparable logic frameworks such as *B*, *Object Constraint Language (OCL)*, *VDM* and *Z* are based on the same logic concept, but use different techniques in the background [24]. While Alloy uses SAT solving the other logic frameworks are based on finite state machines. However, utilizing finite state machines leads to an exponential state explosion. Hence, they suffer from the same problem of exponential blowout as the SAT solving strategy of Alloy. However, Alloy has an advantage over the other frameworks. While other frameworks cannot open up the solution space automatically and require a manually defined test specification, Alloy is based on a full automatic analysis that does not analyse the complete solution space [24]. This technique is called *constraint solving* and offers fast feedback for limited solution spaces. Since the look-ahead window represents such a limited solution space, Alloy meets the defined requirements better than other frameworks. This strategy possesses additionally advantages against approaches that would use a more expressive formal language than first-order logic. For example, the Alloy language is less expressive than regular expression, because it is limited to first-order logic. However, using regular expressions requires separate symbols for each possible state of the process. This encompasses the order of activities as well as values of variables and other resources. To avoid a possibly unlimited number of states, it becomes again necessary to limit the solution space. Furthermore, there exists currently no approach that transforms a multi-perspective declarative process model to regular expressions (only for single-perspective languages like Declare). Hence, Alloy seems to be an adequate solution as demonstrated by the successful use for the simulation and the execution of multi-perspective declarative process models [2,3].

However, the limited expressiveness compared to formal languages such as regular expressions could be a potential problem. Nevertheless, the expressiveness is sufficient to cover modelling languages like DPIL and MP-Declare as seen in the previous sections. Technically, the expressiveness of these languages is an open question in research and not theoretically proven; however, until now the provided expressiveness is sufficient to cover all element of these modelling languages. For extending the approach to other declarative modelling languages, e.g. to DCR graphs, it must be evaluated whether the expressiveness of Alloy logic is sufficient.

Apart from the technical points, various studies consider Alloy to be useful in similar areas as the application areas of the look-ahead. In [40], Alloy is described as useful for testing and verifying code in software projects. The authors emphasize the good balance between analysability and expressiveness of Alloy as well as the availability of the important modelling tasks simulation, verification and debugging. The ability for incremental building and exploration of system design are very similar to modelling declarative process models in order to avoid an under- or overspecification.

4.5 Applications for the look-ahead approach

We conclude this section by describing two promising applications for our introduced look-ahead functionality: predictive business process monitoring and process modelling or checking.

A wide look-ahead helps to take appropriate measures at an early stage. The revealed consequences of the look-ahead can be combined with domain knowledge of the particular process to give recommendations to the process participants, e.g. a very cost and time saving execution path or the abort

of the process execution. It also becomes possible to predict the most likely process trajectory by comparing the current execution with historic process executions.

Process models are checked by comparing the desired behaviour with the expected behaviour provided in the model [8,42]. For this task, exemplary process executions in form of traces are used. In context of process modelling each trace describes a scenario, i.e. a specific situation of the process. The look-ahead allows to analyse such scenarios by determining possible consequences. The resulting consequences provide information, whether the process model is conformant with reality and reveal undesired behaviour like contradictions, deadlocks or undesired process trajectories. Studies in cognitive science confirm the positive effects of examples especially of counterexamples for understanding [43].

5 Implementation and evaluation

In this section, we provide implementation details and evaluate the proposed look-ahead concept by the running example and a further multi-perspective declarative process model. Afterwards, we emphasize the difference to the previous approach presented in [25], check which of the requirements defined in Sect. 3.2 are fulfilled, and discuss existing limitations.

5.1 Implementation

The proposed look-ahead approach has been implemented as a Java application including a graphical user interface (cf. Figs. 6, 7). The graphical user interface allows the definition of involved process entities (cf. Fig. 6(1)), the import of a DPIL model or an MP-Declare model (cf. Fig. 6(2)), and the definition of the current state of the process execution (cf. Fig. 7(3)). The user has the possibility to define the intended next step activity (cf. Fig. 7(4)) as well as the look-ahead length (cf. Fig. 7(5)). By clicking the configuration button (cf. Fig. 7(7)) for counterexamples, the user can define the search for counterexamples in detail (cf. Sect. 4.2). By clicking on the run button (cf. Fig. 7(6)), the inputs are transformed to Alloy code, which is directly executed via the JavaAPI[4] provided by Alloy.

Alloy ships with different SAT solvers, which are interchangeable due to the multiple abstraction layers in the architecture of Alloy. The set of available SAT solvers depends on the platform, since some of them require a 32bit execution environment or a linux distribution. The SAT solvers offer some configuration options like space allocation or symmetry breaking. Symmetry breaking prevents symmetric solutions, i.e. solutions that are isomorphic [24]. In our implementation, we always enable symmetry breaking and allocate the maximum of available space.

A fixed number of generated instances are shown as a preview (cf. Fig. 7(8)). All generated instances can be exported for further processing. Therefore, generated instances by the SAT solver are parsed and transformed into the well-known XES format utilizing the *OpenXES* library.[5]

5.2 Evaluation

SAT solving for propositional logic is known to be NP-complete (Cook–Levin theorem [12]). Hence, it is important to evaluate the look-ahead that is based on this principle in term of its runtime performance. Note, that measuring the runtime performance does not primarily evaluate the look-ahead itself rather than the used SAT solver. For evaluation we use the single-perspective process model hiddenCoExistence from the introduction, the running example, and an additional process model we describe shortly in the following.

Example 2 The *realEstate* process model describes the process of visiting and buying a real estate on an abstract level. Its complexity, measured in the number of constraints, process entities and domain of the involved data objects, is less than of the *publishPaper* process. We describe this process model in the same manner as the running example. The control flow is described in Fig. 8. The hereby used constraints mean that:

1. before buying a real estate, the customer has to request an application,
2. if an application was requested, it would have to be handled by a real estate agent,
3. before buying a real estate the application has to be approved,
4. the customer has to visit the real estate before buying.

Furthermore, we have the following cross-perspective constraints:

1. customer and real estate agent should not be the same,
2. the incomes are of four categories. Only if the category is 1 or 4 it is possible for the customer to buy the real estate.

We run this process with five different persons.

We identified the following influencing factors on the computational time: *(i)* the number of process entities, *(ii)* the

[4] http://alloytools.org/documentation/alloy-api/index.html.

[5] http://www.xes-standard.org/openxes/start.

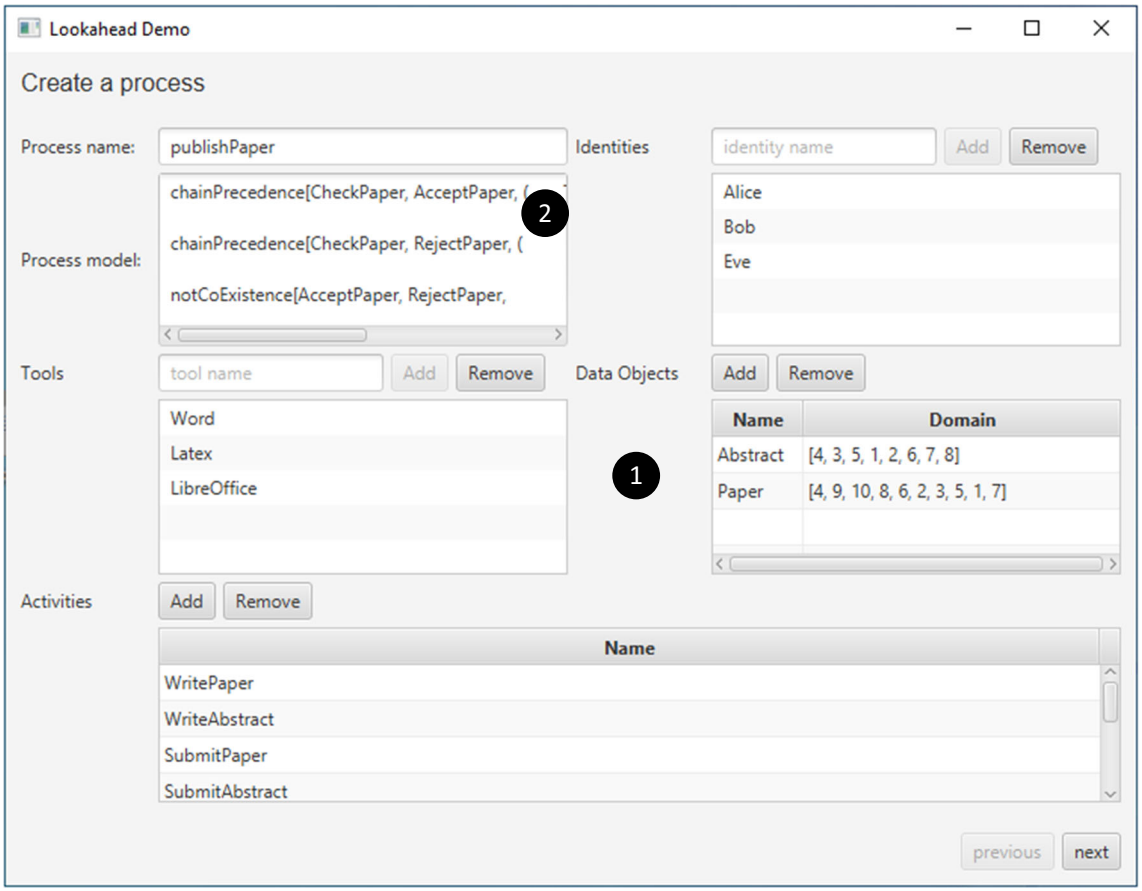

Fig. 6 Graphical User Interface of the look-ahead Demo

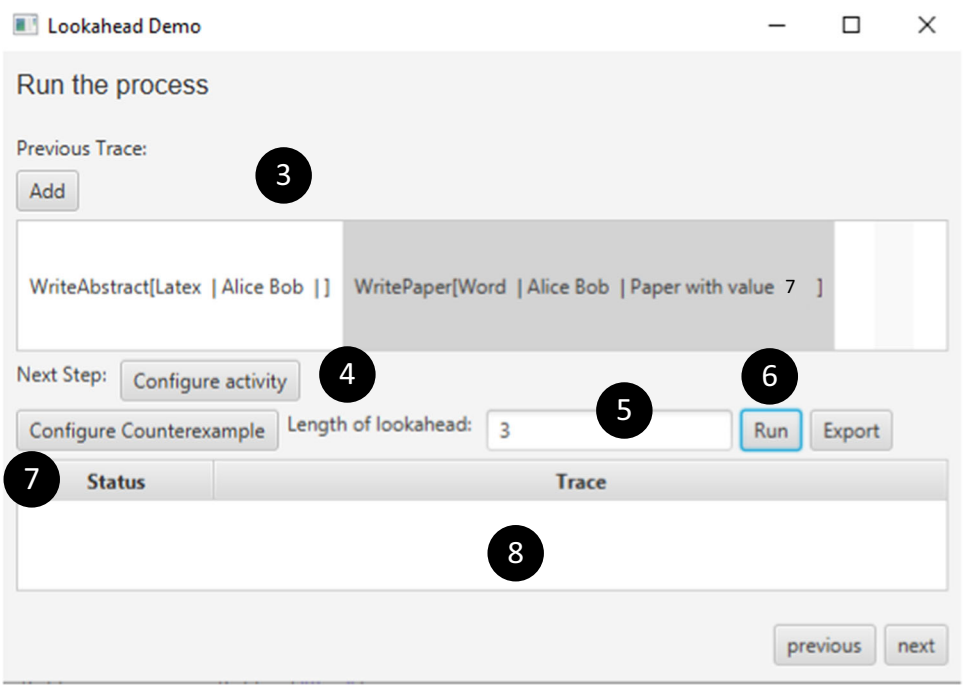

Fig. 7 Graphical User Interface for defining previous trace and look-ahead

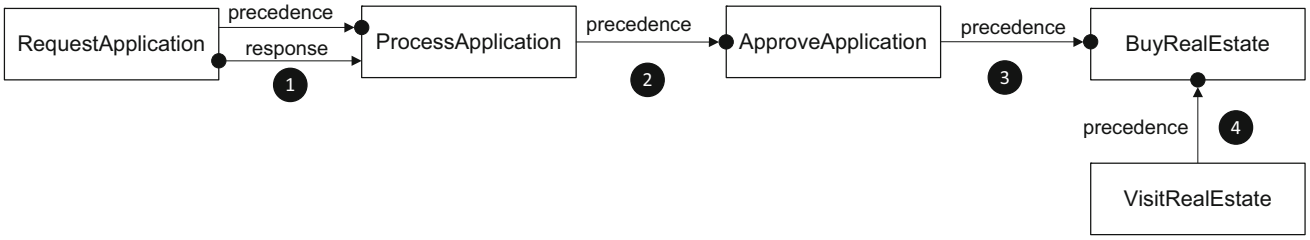

Fig. 8 *realEstate* example process in graphical Declare notation

number of constraints the model contains and *(iii)* the length of the trace. The influence of these factors was already evaluated in [3] and the trace length was revealed as the driving influencing factor. These results hold for the look-ahead, too. In our analysis, we focussed on two further points. First, we investigate how the different process perspectives affect the runtime. Therefore, we analyse the runtime separately for different process perspectives, i.e. we iteratively omit process perspectives. Second, we compare the performance of two SAT solvers (SAT4J and MiniSAT). All benchmarks have been performed on a Windows 10 64 Bit system equipped with an Intel Core i7-4790K CPU @4.00GHZ 16GB memory and SSD drive.

In the following, we analyse the runtime performance of our measures.

hiddenCoExistence In runtime analysis of the single-perspective *hiddenCoExistence* process model (cf. Table 1), we observe an exponential growing runtime in dependent on the look-ahead length. Also the above mentioned dependency on the trace length can be observed. In case of invalid previous traces (marked as bold) the runtime increases only slowly. The runtime of the used SAT solvers (SAT4J and MiniSAT) are nearly identical, which is why we present only the results of the SAT4J solver in Table 1.

publishPaper and realEstate The runtime analysis for the *publishPaper* process model and the *realEstate* process model are shown in Tables 3 and 2, respectively. In this analysis, we are focussing on the influence of different process perspectives and the SAT solvers. We observe that the data-oriented perspective affects the runtime most, while the control-flow perspective has a slight influence only. The operational perspective and organizational perspective have a comparable effect on the runtime. Reasons for the difference to the data-oriented perspective can be traced back to the domain of the data-objects. With respect to the large runtime, the difference between the used SAT solvers is emphasized more clearly as in the *hiddenCoExistence* process model. In both analyses, we neglect a deeper analysis of the influence of the trace length, since the effect is already investigated in [3].

In summary, the presented approach outperforms the previous one in [25]. However, the runtime currently limits the approach to rather small application scenarios or scenarios with less strict time requirements, like model checking. Due to the interchangeability of the SAT solvers in Alloy, our future work will include an extensive study of different SAT solvers that provide better parallelization than the currently used SAT solvers.

5.3 Checking the fulfillment of the requirements

Next, we give a brief summary which of the requirements for the look-ahead functionality are met by our implementation. Note, that the requirements are defined in Sect. 3.2 . Requirement 1 (Completeness of simulation) is fully met, since Alloy guarantees distinctness and exhaustiveness while generating solutions. Also redundant solutions are avoided through the Alloy Analyzer. The second requirement (Systematic and efficient search) is largely fulfilled within the possibilities of a SAT solving-based approach. The used SAT solvers use different strategies to search the solution space in an efficient manner. However, the current runtime limits the approach to rather small application scenarios or scenarios with less strict time requirements. Requirement 3 (Language independence) is fully achieved through the support of the modelling languages Declare, MP-Declare and DPIL. Furthermore, an extension to other modelling languages appears realistic, even if a final check of the expressiveness of the underlying logic concepts has to be checked. Also the last requirement (Output format for easy post-processing) is achieved, since nearly every process mining tool is built upon the chosen XES format. Hence, an easy post-processing of the generated traces is possible.

5.4 Difference to the previous approach

We conclude this section by describing the difference to the previous approach in [25]. The main difference is the way how we use the SAT solving capabilities of Alloy. In [25] a brute-force algorithm generates all possible process trajectories and transforms them into Alloy code as described in Sect. 4.1.3. Afterwards, the Alloy Analyzer is used to check these traces for validity. Therefore, the Alloy Analyzer has to search for a solution in the given scope. The restrictions

Table 1 Runtime analysis of the single-perspective *hiddenCoExistence* process model

	Look-ahead length									
Prev. Trace	1	2	3	4	5	6	7	8	9	10
$\langle\rangle$	0s	0s	0s	0s	1s	2s	7s	22s	68s	239s
$\langle A\rangle$	0s	0s	0s	0s	1s	3s	8s	26s	83s	320s
$\langle B\rangle$	0s	0s	0s	1s	2s	5s	15s	46s	158s	621s
⟨C⟩	**0s**	**0s**	**0s**	**0s**	**0s**	**0s**	**0s**	**0s**	**0s**	**0s**
$\langle D\rangle$	0s	0s	0s	0s	1s	1s	3s	8s	10s	23s
$\langle E\rangle$	0s	0s	0s	0s	1s	2s	3s	8s	10s	20s
$\langle A, B\rangle$	0s	0s	0s	1s	2s	6s	16s	61s	148s	463s
⟨A, C⟩	**0s**	**0s**	**0s**	**0s**	**0s**	**1s**	**2s**	**3s**	**5s**	**5s**
$\langle A, B, C\rangle$	0s	0s	0s	1s	3s	7s	21s	72s	240s	797s
⟨A, B, D⟩	**0s**	**0s**	**0s**	**0s**	**1s**	**2s**	**3s**	**3s**	**5s**	**6s**

Table 2 Runtime analysis of the *realEstate* process model

	Look-ahead length SAT4J					MiniSAT				
Configuration, Trace	1	2	3	4	5	1	2	3	4	5
all persp.,⟨⟩	0s	4s	3m	36m	3.5h	0s	3s	2m	28m	3h
without org. persp.,⟨⟩	0s	0s	6s	5m	42m	0s	0s	4s	4m	33m
without data persp.,⟨⟩	0s	0s	2s	2m	34m	0s	0s	1s	2m	24m
as single persp.,⟨⟩	0s	0s	0s	0s	0s	0s	0s	0s	0s	0s

Table 3 Runtime analysis of the *publishPaper* process model

	Look-ahead length SAT4J					MiniSAT				
Configuration, Trace	1	2	3	4	5	1	2	3	4	5
all persp.,⟨⟩	1s	2m	4h	-	-	1s	2m	3.5h	-	-
without org.,⟨⟩	0s	1m	32m	4h	-	0s	48s	25m	3.5h	-
without op.,⟨⟩	0s	1m	30m	4h	-	0s	45s	24m	3.5	-
without data,⟨⟩	0s	2s	29s	5m	.5h	0s	1s	28s	4m	.5h
as single persp.,⟨⟩	0s	0s	0s	1s	1s	0s	0s	1s	1s	1s

through the process model and the trace leave only two possible solutions: If the trace in the model was valid, the Alloy Analyzer produces exactly one solution which is identical to the trace in the Alloy specification. Otherwise, no solution was found, and the trace was classified as invalid. Hence, for each trace that should be validated the Alloy Analyzer has to search through the whole solution space. Due to the large number of traces that has to be checked, the solution space is searched through many times. In the approach presented in the paper at hand, we generate only valid traces (or invalid traces in case of counterexamples) instead of checking a trace for validity. This has the advantage that the solution space has to be searched through only once which is the reason for the runtime improvement.

6 Related work

This work relates to the stream of research on (multi-perspective) declarative process management. Since the look-ahead approach touches many different aspects, we divide this section into paragraphs with respect to these aspects.

Problems of understanding and modelling of declarative process models The problems in understanding and modelling of declarative process models described at the beginning of this paper are well known in current research. Nevertheless, there are likewise little studies that explicitly deal with the comprehensibility of declarative process models. In [21], the authors present a systematic study focusing on the compre-

hension of Declare models. Their study revealed that single constraints can be handled well by the most subjects, while combinations of constraints pose a significant challenge [21]. The study also showed that subjects use the layout (i.e. the composition of notation elements in the process model) in the process model for reading a Declare model. Hence, the graphical notation can hamper the understanding of the model. Comparable understandability studies have been considered hybrid representations where graphical and text-based specifications of declarative processes have been studied [1,6]. However, currently there exists no comparable study for MP-Declare models.

Hidden dependencies One reason for the difficult understanding of declarative process models are hidden dependencies. The detection of hidden dependencies is addressed in [37]. In [14] hidden dependencies between constraints are identified as one of the main causes, why declarative process models are more difficult to comprehend and require a higher mental effort of both the modeller and the reader. The authors proposed a methodology to detect these hidden dependencies and make them visible. They integrated this method into the Declare Execution Environment,[6] that enriches existing Declare models with visual and textual annotations to clarify allowed or disallowed behaviour by the model. This can be considered in some way as a simplified single-perspective look-ahead strategy. In an empirical study with 95 novice Declare modellers, the practical use of this approach was proven [14].

Redundant constraints and contradictions A very similar problem are redundant constraints and contradictions between constraints. This issue is addressed in [11]. The authors use automata-product monoids to reveal redundancies and inconsistencies. However, this approach is also limited to single-perspective process models.

Execution of multi-perspective declarative process models There are a few approaches for the execution of multi-perspective declarative process models. In [3], the authors transform MP-Declare process models into the logic language Alloy and use the Alloy framework for their execution. The same was done in [2] for DPIL process models. Additionally, there exists an execution framework for DPIL [44]. Also the *DCR graph* framework [22] offers the execution of the multi-perspective declarative process models based on the notion graph marking [35,38]. A further common declarative modelling language is the *Guard-Stage-Milestone (GSM) language* that enables the design of business artefacts with declarative elements to describe the intended behaviour without an explicit definition of the control flow [7,26]. This language is also executable and was embedded into a multi-agent systems semantic for reasoning about knowledge and time at the formalism of first-order logic [7]. In the *EM-BrA^{2}CE* project [19], the *Semantics of Business Vocabulary and Business Rules (SBVR)* framework is extended by concepts such as activities, states and participants. The SBVR rules are translated to event-condition-action (ECA) rules to enable their execution. For some other declarative modelling languages, such as *CLIMB* [27], there exist no frameworks for modelling and execution.

Mining of multi-perspective declarative process models The automatic extraction of multi-perspective declarative process models has been addressed in latest research as well. In [32], the authors present an approach for the extraction of MP-Declare models based on SQL queries. A similar approach based on parallel and distributed computing is presented in [39]. A toolkit and software framework for deriving MP-Declare rules is given in [5].

Generating process execution traces The generation of traces and, as a result, of complete event logs gains more and more attention in research in the last years, since generated event logs are often better suited for evaluating various process mining algorithms than real-life event logs [2,4,34]. While a large number of trace generators already exist for imperative process models, there exists only a small number of generators for (multi-perspective) declarative process models. The existing generators are mostly limited to the control flow perspective and organizational perspective. So far, only two approaches provide additional support for the data-oriented perspective. In [34], the authors present an Alloy-based event log generator for MP-Declare process models and in [2] a trace generator for DPIL process models is presented. However, both approaches are limited to the behavioural, organizational and data-oriented perspective. If a sufficient number of completed process executions are already available, Long Short-Term Memory Neural Networks (LSTMN networks), such as in [10], can be trained to simulate further process trajectories or entire event logs. However, such approaches depend on a sufficient number of already completed traces and may generate process trajectories that are not conformant to the underlying process model. Furthermore, the approach is currently limited to the control flow perspective, organizational perspective and time perspective.

Predictive business process monitoring Predictive business process monitoring is an extensively discussed topic in research. However, existing approaches are mostly limited to control-flow, organizational and time perspective. In general, those approaches are based on event logs, so they can be also applied to knowledge-intensive processes. The proposed look-ahead in this paper is also used for prediction tasks; however, there is a main difference to common predictive business process monitoring techniques like neural networks [10,33]. While neural networks are trained on completed process trajectories and predict the most likely execution, the look-ahead returns based on a process model, all possible process trajectories up to a certain length [25]. However, trained

[6] http://www.processmining.be/declareexecutionenvironment/.

models work statistical and are consequently error prone in this way, that they can predict non-compliant process trajectories. In contrast, the logic-based look-ahead predicts only conformant process trajectories. However, the look-ahead does not consider experience gained in training data.

7 Conclusion and future work

In this paper, we introduced a logic-based look-ahead approach for multi-perspective declarative process models. Specifically, we contributed (1) an overview over challenges in the simulation and validation of declarative process models; (2) a realization of a look-ahead functionality for two declarative process modelling languages (MP-Declare and DPIL) that can be extended to further languages in the future; and (3) an experimental investigation of the look-ahead approach including a discussion of strengths and limitations.

In future work, the approach needs a more rigorous evaluation that investigates different SAT solvers in depth. Since the SAT problem is NP-complete the focus should primarily be on parallelization to achieve a shorter runtime. The development of hybrid process modelling techniques also affects execution and simulation approaches like the proposed look-ahead. Hence, the idea of a look-ahead should also be transferred to hybrid process models. Furthermore, the underlying meta-model should be extended to support further process perspectives, especially the time perspective, without using the data-oriented perspective as a workaround. Additionally, the lifecycle of activities and various types of events, such as human tasks, service tasks or automatic tasks, should be integrated in the meta-model. Since there is a variety of different declarative modelling languages the approach should be extended to further languages, especially DCR graphs and CMMN. In this context, an analysis of the expressiveness of common declarative modelling languages would be helpful to investigate the question which formal language should be used to validate, execute and simulate multi-perspective declarative process models.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Abbad Andaloussi, A., Burattin, A., Slaats, T., Petersen, A.C.M., Hildebrandt, T.T., Weber, B.: Exploring the understandability of a hybrid process design artifact based on dcr graphs. In: Reinhartz-Berger, I., Zdravkovic, J., Gulden, J., Schmidt, R. (eds.) Enterprise, Business-Process and Information Systems Modeling, pp. 69–84. Springer, Cham (2019)
2. Ackermann, L., Schönig, S., Jablonski, S.: Simulation of Multi-perspective Declarative Process Models. In: BPM Workshops, pp. 61–73. Springer (2016)
3. Ackermann, L., Schönig, S., Petter, S., Schützenmeier, N., Jablonski, S.: Execution of multi-perspective declarative process models. In: OTM 2018 Conferences, pp. 154–172 (2018)
4. Ackermann, L., Schönig, S.: Mudeps: Multi-perspective declarative process simulation. In: Azevedo, L., Cabanillas, C. (eds.) Proceedings of the BPM Demo Track 2016, CEUR Workshop Proceedings, vol. 1789, pp. 12–16 (2016)
5. Alman, A., Ciccio, C.D., Haas, D., Maggi, F.M., Mendling, J.: Rule mining in action: the rum toolkit. In: Ciccio, C.D., Depaire, B., Weerdt, J.D., Francescomarino, C.D., Munoz-Gama, J. (eds.) Proceedings of the ICPM Doctoral Consortium and Tool Demonstration Track, CEUR Workshop Proceedings, vol. 2703, pp. 51–54
6. Andaloussi, A.A., Buch-Lorentsen, J., L\'pez, H.A., Slaats, T., Weber, B.: Exploring the modeling of declarative processes using a hybrid approach. In: Conceptual Modeling—38th International Conference, ER 2019, Salvador, Brazil, November 4-7, 2019, Proceedings, pp. 162–170. Springer (2019). https://www.alexandria.unisg.ch/258707/
7. Belardinelli, F., Lomuscio, A., Patrizi, F.: Verification of gsm-based artifact-centric systems through finite abstraction. In: Proceedings of the 10th International Conference on Service-Oriented Computing, ICSOC'12, pp. 17–31. Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34321-6_2
8. Burattin, A., Maggi, F.M., Sperduti, A.: Conformance checking based on multi-perspective declarative process models. Expert Syst. Appl. **65**, 194–211 (2016)
9. Bussler, C.: Analysis of the organization modeling capability of workflow-management-systems. In: PRIISM'96 Conference Proceedings, pp. 438–455 (1996)
10. Camargo, M., Dumas, M., González-Rojas, O.: Learning accurate lstm models of business processes. In: Hildebrandt, T., van Dongen, B.F., Röglinger, M., Mendling, J. (eds.) Business Process Management, pp. 286–302. Springer, Cham (2019)
11. Ciccio, C.D., Maggi, F.M., Montali, M., Mendling, J.: Resolving inconsistencies and redundancies in declarative process models. Inf. Syst. **64**, 425–446 (2017). https://doi.org/10.1016/j.is.2016.09.005
12. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC'71, pp. 151–158. Association for Computing Machinery, New York, NY, USA (1971). https://doi.org/10.1145/800157.805047
13. de Leoni, M., van der Aalst, W.M.P., Dees, M.: A general process mining framework for correlating, predicting and clustering dynamic behavior based on event logs. Inf. Syst. **56**, 235–257 (2016). https://doi.org/10.1016/j.is.2015.07.003
14. De Smedt, J., De Weerdt, J., Serral, E., Vanthienen, J.: Improving understandability of declarative process models by revealing hidden dependencies. In: Nurcan, S., Soffer, P., Bajec, M., Eder, J. (eds.) Advanced Information Systems Engineering, pp. 83–98. Springer, Cham (2016)
15. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer, Cham (2018)

16. Fahland, D., Lübke, D., Mendling, J., Reijers, H., Weber, B., Weidlich, M., Zugal, S.: Declarative versus imperative process modeling languages: the issue of understandability. In: Halpin, T., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Soffer, P., Ukor, R. (eds.) Enterprise, Business-Process and Information Systems Modeling, pp. 353–366. Springer Berlin Heidelberg, Berlin (2009)
17. Fahland, D., Mendling, J., Reijers, H.A., Weber, B., Weidlich, M., Zugal, S.: Declarative versus imperative process modeling languages: the issue of maintainability. In: Rinderle-Ma, S., Sadiq, S., Leymann, F. (eds.) Business Process Management Workshops, pp. 477–488. Springer Berlin Heidelberg, Berlin (2010)
18. Gabbay, D.: The declarative past and imperative future. In: Banieqbal, B., Barringer, H., Pnueli, A. (eds.) Temporal Logic in Specification, pp. 409–448. Springer Berlin Heidelberg, Berlin (1989)
19. Goedertier, S., Haesen, R., Vanthienen, J.: Rule-based business process modelling and enactment. Int. J. Bus. Process Integr. Manag. **3**(3), 194–207 (2008)
20. Group, O.M.: Business process modeling notation version 2.0. technical report, object management group final adopted specification (2011)
21. Haisjackl, C., Barba, I., Zugal, S., Soffer, P., Hadar, I., Reichert, M., Pinggera, J., Weber, B.: Understanding declare models: strategies, pitfalls, empirical results. Softw. Syst. Model. **15**(2), 325–352 (2016). https://doi.org/10.1007/s10270-014-0435-z
22. Hildebrandt, T.T., Mukkamala, R.R., Slaats, T., Zanitti, F.: Contracts for cross-organizational workflows as timed dynamic condition response graphs. J. Log. Algebr. Program. **82**(5–7), 164–185 (2013). https://doi.org/10.1016/j.jlap.2013.05.005
23. Hull, R., Damaggio, E., De Masellis, R., Fournier, F., Gupta, M., Heath, F.T., Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P.N., Vaculin, R.: Business artifacts with guard-stage-milestone lifecycles: Managing artifact interactions with conditions and events. In: Proceedings of the 5th ACM International Conference on Distributed Event-Based System, DEBS'11, pp. 51–62. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2002259.2002270
24. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2012)
25. Käppel, M., Schützenmeier, N., Schönig, S., Ackermann, L., Jablonski, S.: Logic based look-ahead for the execution of multi-perspective declarative processes. In: Reinhartz-Berger, I., Zdravkovic, J., Gulden, J., Schmidt, R. (eds.) Enterprise, Business-Process and Information Systems Modeling, pp. 53–68. Springer, Cham (2019)
26. Lomuscio, A., Griesmayer, A., Gonzalez, P.: Verifying gsm-based business artifacts. In: 2013 IEEE 20th International Conference on Web Services, pp. 25–32. IEEE Computer Society, Los Alamitos, CA, USA (2012). https://doi.org/10.1109/ICWS.2012.31
27. Montali, M.: Specification and Verification of Declarative Open Interaction Models: a Logic-Based Approach. Ph.D. thesis, University of Bologna (2009)
28. Pesic, M.: Constraint-based workflow management systems: shifting control to users. Ph.D. thesis, Industrial Engineering and Innovation Sciences (2008). https://doi.org/10.6100/IR638413. Proefschrift
29. Reichert, M., Weber, B.: Flexibility Issues in Process-Aware Information Systems, pp. 43–55. Springer Berlin Heidelberg, Berlin (2012). https://doi.org/10.1007/978-3-642-30409-5_3
30. Rozinat, A., Mans, R.S., Song, M., van der Aalst, W.M.P.: Discovering simulation models. Inf. Syst. **34**(3), 305–327 (2009). https://doi.org/10.1016/j.is.2008.09.002
31. Schönig, S., Ackermann, L., Jablonski, S.: Towards an implementation of data and resource patterns in constraint-based process models. In: Modelsward, pp. 271–278 (2018)
32. Schönig, S., Ciccio, C.D., Maggi, F.M., Mendling, J.: Discovery of multi-perspective declarative process models. In: Sheng, Q.Z., Stroulia, E., Tata, S., Bhiri, S. (eds.) Service-Oriented Computing—14th International Conference, ICSOC, Lecture Notes in Computer Science, vol. 9936, pp. 87–103
33. Schönig, S., Jasinski, R., Ackermann, L., Jablonski, S.: Deep learning process prediction with discrete and continuous data features. In: Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2018, pp. 314–319. SCITEPRESS - Science and Technology Publications, Lda, Portugal (2018). https://doi.org/10.5220/0006772003140319
34. Skydanienko, V., Francescomarino, C.D., Maggi, F.: A tool for generating event logs from multi-perspective declare models. In: BPM (Demos) (2018)
35. Slaats, T., Mukkamala, R., Hildebrandt, T., Marquard, M.: Exformatics declarative case management workflows as dcr graphs. In: BPM, pp. 339–354 (2013)
36. Smedt, J.D., Weerdt, J.D., Vanthienen, J., Poels, G.: Mixed-paradigm process modeling with intertwined state spaces. Bus. Inf. Syst. Eng. **58**(1), 19–29 (2016). https://doi.org/10.1007/s12599-015-0416-y
37. Smedt, J.D., Weerdt, J.D., Serral, E., Vanthienen, J.: Discovering hidden dependencies in constraint-based declarative process models for improving understandability. Inf. Syst. **74**(Part), 40–52 (2018)
38. Strømsted, R., López, H.A., Debois, S., Marquard, M.: Dynamic evaluation forms using declarative modeling. In: BPM (2018)
39. Sturm, C., Schönig, S., Ciccio, C.D.: Distributed multi-perspective declare discovery. In: Clarisó, R., Leopold, H., Mendling, J., van der Aalst, W.M.P., Kumar, A., Pentland, B.T., Weske, M. (eds.) Proceedings of the BPM Demo Track and BPM Dissertation Award, CEUR Workshop Proceedings, vol. 1920
40. Torlak, E., Taghdiri, M., Dennis, G., Near, J.P.: Applications and extensions of alloy: past, present and future. Math. Struct. Comput. Sci. **23**, 915–933 (2013)
41. van der Aalst, W.: The application of petri nets to workflow management. J. Circuits Syst. Comput. **8**, 21–66 (1998). https://doi.org/10.1142/S0218126698000043
42. van der Aalst, W.M.P.: Process Mining-Discovery, Conformance and Enhancement of Business Processes. Springer Berlin Heidelberg, Wiesbaden (2011)
43. Zazkis, R., Chernoff, E.: What makes a counterexample exemplary? Educ. Stud. Math. **68**, 195–208 (2008). https://doi.org/10.1007/s10649-007-9110-4
44. Zeising, M., Schönig, S., Jablonski, S.: Towards a common platform for the support of routine and agile business processes. In: Collaborative Computing: Networking, Applications and Worksharing (2014)

Martin Käppel is a research assistant with the Institute for Computer Science at University of Bayreuth (Germany). He received the master's degree (with honours) at University of Bayreuth. His research is focused on Process Mining and the development of Small Sample Learning methods for BPM. Based on his work, he published scientific papers in international conferences and journals.

Lars Ackermann is an Assistant Professor of Computer Science with the Institute for Computer Science at University of Bayreuth (Germany). He received the master's degree (with honours) in Computer Science and the doctoral degree from University of Bayreuth. He has an established background in BPM/Process Mining and has been working in this field for several years. He published extensively in the research area of business process management and information systems, both in international conferences and journals.

Stefan Schönig is a Professor for Information Systems with the Institute of Management Information Systems at the University of Regensburg in Germany. He received both the master's degree (with honours) in Applied Computer Science (Engineering/Computer Science) and the doctoral degree from University of Bayreuth. Before, he held a position as a tenured assistant professor at University of Bayreuth. He was a post-doctoral researcher with the Institute for Information Business at WU Vienna (Vienna University of Economics and Business). He has an established background in BPM/Process Mining and IoT research and has been working in this field for over 9 years. He published extensively in the research area of BPM and information systems, both in international conferences and journals.

Stefan Jablonski is a Full Professor of Computer Science with the Institute for Computer Science at University of Bayreuth (Germany). He is head of the chair for Databases and Information Systems. His major research interests include Business Process Management, flexible process enactment technologies and metamodelling. He has been participating in numerous national and international BPM research as well as industrial projects.