Simulation of Multi-Perspective Declarative Process Models

Lars Ackermann, Stefan Schönig and Stefan Jablonski

University of Bayreuth, Germany
{firstname.surname}@uni-bayreuth.de

Abstract Imperative languages like BPMN are eminently suitable for representing routine processes and are likewise cumbersome in case of flexible processes. The latter are easier to describe using declarative process modeling languages (DPMLs). However, understandability and tool support of DPMLs are comparatively poor. Additionally, there may be an affinity to a particular language caused by company guidelines or individual preferences. Hence, a technique for automatically translating process models between different languages is required. Process models usually describe several aspects of a process, such as activity orderings and role assignments. Therefore, our approach focuses on translating resource-aware process models. We utilize well-established techniques for process simulation and mining to avoid the definition of cumbersome model transformation rules. Since there is currently no simulation approach for resource-aware declarative process models we additionally describe a solution based on a well-known and expressive logic. The whole translation approach is discussed and evaluated at the example of BPMN and DPIL.

Keywords: process model translation, simulation, process mining

1 Introduction

An important task in business process management is the creation of business process models in order to provide more or less detailed insights into the underlying process. Business process models are usually created manually using dedicated modeling tools. However, in many cases knowledge about previous process executions is already available, e.g. in the form of event logs. Consequently, techniques have been developed that are able to extract and analyze the contained information or even provide the opportunity to *discover* a process model that is able to describe the observed data. The latter techniques are artifacts of a rising discipline called *process mining*.

A recent standard for storing and analyzing such event logs is the *eXtensible Event Stream (XES)* [1] format. The standard stipulates that an event corresponds to the execution of exactly one *activity* whereby an activity denotes a well-defined step in the process it is part of [2]. An event belongs to exactly one *case* which, in general, is usually called process *trace* and describes one particular process instance. The events in one trace are ordered chronologically, either

explicitly, e.g. by their actual order in the log, or implicitly using time stamps. An event in the log may also consist of additional information about involved organizational resources, data and operations.

Usually information about former process executions are stored in a form that does not coincide with the XES or any other log standard. Information can be scattered over several sub-systems and representations and, therefore, must be extracted and transformed in order to apply process mining algorithms. Though this data extraction and transformation is an integral part of process mining approaches, it can be clearly separated from the process discovery part. In order to evaluate the quality of the discovery approach test data, i.e. event logs are needed. Producing logs manually is a time-consuming task and real-life logs are rare[1] usually suffer from infrequent behavior (*noise*) that is not representative for executions of the underlying process. Another issue called *incompleteness* denotes that lack of information and is mostly encountered when using event logs that are too small. Both can affect the process discovery in an inexplicable way. Hence, commonly process discovery approaches are evaluated based on *synthetic* event logs that are generated by simulation tools [3]. An additional major advantage of using simulation is, that the researcher can influence the characteristics of the generated event logs, e.g. in terms of length and contents.

Simulation tools are usually tailored to a particular process modeling language, which is why a variety of different techniques have been developed (???Refs einfuegen). As already mentioned in [3] all of these techniques simulate imperative models such as *BPMN* [4] models or *Petri Nets* [5]. Over the last years, *declarative* process modeling languages and, consequently, declarative process discovery techniques gained more and more attraction [6–13]. Imperative languages model the underlying process *explicitly* using flow-oriented representations and based on a closed-world assumption. A flow of activities that is not in the model is implicitly forbidden. In contrast, declarative languages assume an open world and arbitrary process executions which can be restricted using constraints. Due to this semantic gap, simulation techniques for imperative models are not suitable for declarative models [3]. Consequently, there is a lack of simulation tools for declarative models. The approach presented in [3] is the only representative and is able to generate traces based on rules that restrict the temporal ordering and the existence of activities in the particular process. However, the simulator as well as the modeling language it is based on does not consider other process perspectives, like the organizational or the data-oriented perspective [14, 15]. Since flexible processes, which are usually modeled using declarative languages, are often *human-centric* [7] both a more expressive modeling language and a corresponding simulation technique are required. Hence, the approach presented in the paper at hand is designed to complement the plain control-flow-based simulation with a simulation tool, which is able to handle *multi-perspective* process models and primarily focuses on the organizational and the data-oriented process perspectives. Furthermore, it is based on the multi-

[1] Some sets of event logs have been provided within the scope of the *BPI Challenges* since 2011: `http://www.win.tue.nl/bpi`

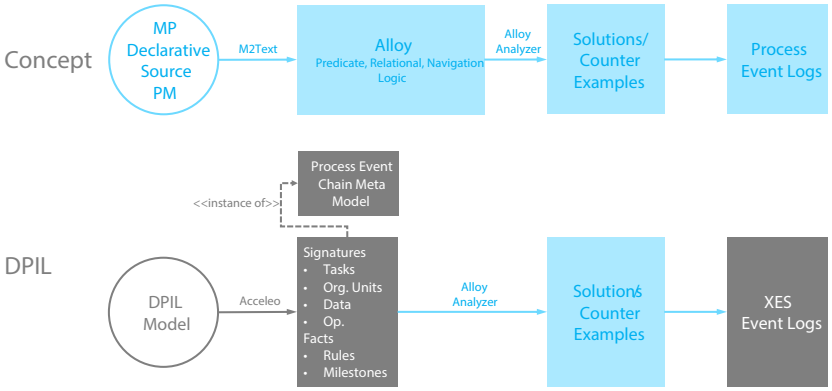

Fig. 1: Concept of Multi-Perspective Declarative Process Model Simulation

perspective *Declarative Process Intermediate Language (DPIL)* [11] (Fig. 1). The approach presented in [3] uses *Regular Expressions* to represent the process rules. Regular expressions are one-dimensional and are, therefore, not suitable to represent process rules that restrict multiple dimensions of the particular process. Hence, we based our simulation technique on a transformation of DPIL rules to a logic language called *Alloy* [16] which was originally used for describing the structure of software systems. Alloy ships with an analyzer that is able to produce examples and counter examples for a given Alloy model. It is possible to configure the simulator in order to produce logs with desired characteristics like the size, the maximum trace length, the trace contents or relative to a partial process execution history.

The remaining part of the paper is structured as follows: ...

2 Background

In this section we introduce the core building blocks of our approach, i.e., the declarative process modelling paradigm and DPIL as a language, process simulation and process mining.

2.1 Multi-Perspective Declarative Process Modeling with DPIL

Research has shown that DPMLs are able to cope with a high degree of flexibility [7]. The basic idea is that, without modeling anything, everything is allowed. To restrict this maximum flexibility, DPMLs allow for formulating constraints which form a forbidden region. Independent from a specific modelling paradigm different perspectives on a process exist. The organizational perspective deals with the definition and the allocation of human and non-human resources to activities. The *Declarative Process Intermediate Language (DPIL)* [11] is a declarative process modelling language that is, unlike other declarative languages multi-perspective, i.e., it allows for representing several business process perspectives,

namely, control flow, data and especially resources. The expressiveness of DPIL and its suitability for business process modelling have been evaluated [11] with respect to the well-known Workflow Patterns. DPIL provides a textual notation based on the use of *macros* to define reusable rules, as it is shown in Fig. **??** (b) and (d). For instance, the *sequence* macro (e.g. $sequence(A, B)$) states that the existence of an *execution event* of task b implies the previous occurrence of an execution event of task A; and the *binding* macro ($binding(A, B)$) states that an activity B is assigned to the same actor that already performed activity A.

Furthermore it is possible to assign an activity directly to a certain role using the *role* macro (e.g. $role(A, R1)$). In example Fig. **??** (a) each activity is executable at most once. In DPIL this can be modeled via the *once* macro (e.g. $once(A)$). In order to specify process goals the language provides the *milestone* keyword. A process instance cannot be completed until all milestone conditions are fulfilled.

Macro	**Expanded Pattern**	**Semantic**
sequence(a,b)	event(of b at :t) implies event(of a at $< t$)	Task b cannot be started before task a has been completed.
once(a)	event(of a at :p) implies not event(of a at $< p$)	The task a only can be started if it was not previously completed once already.
consumes(c,i)	event(of c at :t) implies write(of i at $< t$)	The task c can not be started before a value for the data object i is present.
produces(p,o)	event(of p :t) implies write(of o at $< t$)	The task p can not be completed before a value for the data object o is present.
role(a,id)	event(of a by :id) implies relation(subject id predicate $hasRole$ object r)	The task a must be performed by a process participant having the role id.
binding(a,b)	event(of b by :id) implies event(of a by id)	The tasks a and b must be processed by the same identity.

Table 1: Basic set of multi-perspective macros of the DPIL language

2.2 Running Example Process in DPIL

Fig. 1 shows a model of a simple process for trip management in DPIL. It states that it is mandatory for all applicants to produce the application document for a business trip before it can be approved (*produces* and (*consumes*)). Flights and accommodations can only be booked after the application has been approved (*sequence*). Every task can only be performend once (*once*). The task *Approve application* must be performed by a resource in role *Administration*. Since the applicant usually knows appointments best it is recommended that she books the flight and the accommodation herself (*binding*). A process instance is finished when both flights and accommodations have been booked (*milestone*).

```
use group Administration

process BusinessTrip {
    task Apply for Trip
    task Approve application
    task Book flights
    task Book accommodations

    document Application

    // Constraints of Data Perspective
    ensure produces(Apply for Trip, Application)
    ensure consumes(Approve application, Application)

    // Constraints of Control-Flow Perspective
    ensure sequence(Approve application, Book flights)
    ensure sequence(Approve application, Book accommodations)
    ensure once(Apply for Trip)
    ensure once(Approve application)
    ensure once(Book flights)
    ensure once(Book accommodations)

    // Constraints of Organisational Perspective
    ensure role(Approve Application, Administration)
    ensure binding(Book flight, Apply for Trip)
    ensure binding(Book accommodation, Apply for Trip)

    milestone "Done": event(of Book flights) and event(of Book accommodations)
}
```

Figure 1: Process for trip management modelled with DPIL

2.3 Process Simulation

Business process simulation is usually used for the investigation of correlations in and properties of business processes without resorting to an expensive and in many cases time-consuming observation of real-life process executions [17]. Instead of an exhaustive cost analysis in the paper at hand we refer to the generation of event logs only. Event logs can be generated based on the principle of *Discrete-event Simulation (DES)* [18] which assumes that all state changes of an observed system can be expressed as a discrete sequence of events. Furthermore, process mining techniques are built upon the same premise.

2.4 Alloy in a Nutshell

Alloy is a language for building models that describe structures with respect to desired restrictions. The building bricks for a structure definition are *atoms* and *relations*. Since atoms are indivisible composite structures only can be formed through relations between these atoms. A relation is a set of tuples that consist of a sequence of atoms. It is possible to compose new sets based on well-known standard set operators, such as *union* or *difference*. Some model restrictions cannot be formulated using plain structure descriptions. Therefore, it is possible to add constraints, which can be created using quantification and logical operators. Alloy also comprises a modeling language which is, on the one hand, conventional

enough to support basic programming concepts like modules, polymorphism and parameterized functions. On the other hand it provides new concepts like *scopes*. The scope is important for the *Alloy Analyzer* which is responsible for finding *solutions* that fulfill the Alloy model. The scope limits the maximum size for a solution and enables the analyzer to guarantee an exhaustive instance search. To stick to the paper's focus we only provide a concise description of Alloy's language features if needed.

TODO: Signatures, Facts und Functions erklaeren
- A signature introduces a set of atoms but is similar to a class in object-oriented programming languages (OOPLs), too.

3 Simulation of DPIL with Alloy

3.1 Process Event Chain Meta-Model

Our approach currently focuses on three process perspectives which describes *(i)* the temporal and existential relations between tasks (functional and behavioral perspective), *(ii)* the involvement of resources (organizational perspective), and *(iii)* data dependencies (data perspective). Due to this limited scope we are able to treat activity executions as atomic and, therefore, do not have to take into account the usual activity lifecycle [4]. In Alloy we defined our meta model for PCEs in three modules whereby two are shown in Lst. 1.1 and Lst. 1.2. Both of them are based on another module providing only one signature, called **sig** *AssociatedElement*{}. This signature serves as an interface for extending the meta-model with additional process elements like variables or even elements of new perspectives like operations.

```
module orgmetamodel
open processEventChain_commons

abstract sig Relation {
        subject: one Element,
        object: one Element,
        predicate: one RelationType
}

abstract sig Element extends AssociatedElement {}
abstract sig Identity extends Element{}
abstract sig Group{} extends Element{}
abstract sig RelationType{}
```

Listing 1.1: Organizational Meta-model

Lst. 1.1 is the Alloy implementation of the well known organizational meta-model introduced in [19]. The first line defines the module name. Afterwards, we make the mentioned *AssociatedElement* available by opening the containing module. Line 4-8 allows for the definition of hierarchically structured relations process resources [4] may be involved in based on a *subject-predicate-object-notation*. An example would be: *John (subject) hasRole (predicate) Admin (object)*. In our corresponding Alloy-based process model we need four additional

signatures in order to represent this relation - one for *Relation* itself and one for each of the contained fields.

```
module processEventChain_noLifecycle_multiperspect_IntBased_new

open processEventChain_commons
open orgmetamodel

// Signatures: Process Chain Element Structure
abstract sig PEvent { pos: disj Int }

one sig StartEvent extends PEvent{}{}
one sig EndEvent extends PEvent{}{}

abstract sig TaskEvent extends PEvent { assoEl: some AssociatedElement }{
        #(Task & assoEl) = 1
}
sig HumanTaskEvent extends TaskEvent{}{
        #(Identity & assoEl) = 1
}

abstract sig Task extends AssociatedElement{}

abstract sig DataObject {}
abstract sig DataAccess extends AssociatedElement{
        data: one DataObject
}
abstract sig WriteAccess extends DataAccess{}

// Facts: Additonal non-structural constraints
fact { ∀ intVal: Int • intVal ≥ StartEvent.pos }
fact { ∀ e: (PEvent - StartEvent - EndEvent) •
        e.pos < (StartEvent.pos + #TaskEvent + 1) }
fact { EndEvent.pos ≤ (StartEvent.pos + #TaskEvent + 1) }
fact { ∀ assoEls: (AssociatedElement - Group) •
        assoEls in TaskEvent.assoEl }
fact { ∀ do: DataObject • do in DataAccess.data }
fact { ∀ te: TaskEvent • #(te.assoEl & Group) = 0 }

// Utility Functions
fun exist(asso: AssociatedElement): set TaskEvent {
  { te: TaskEvent • asso in te.assoEl }
}
fun inBefore(curE: TaskEvent, asso: AssociatedElement): set TaskEvent {
  { te: TaskEvent • te.pos < curE.pos and asso in te.assoEl }
}
fun roleOf(id:Identity) : set Group{
  { g: Group • some r: Relation • r.subject=id and r.object in Group }
}
fun dAccess(d: DataObject, type:DataAccess): one DataAccess {
  { da: DataAccess • da in type and d in da.data }
}
```

Listing 1.2: Process Event Chain Meta Model

The structure of process execution event chains (or traces) is described in Lst. 1.2. After defining the module name we make the two previously described modules available (line 3 and 4). The lines 6-25 describe the structural and the remaining lines describe the non-structural properties of a process event chain. From the perspective of object-oriented programming *PEvent* is an abstract class for a general discrete event in the process even chain, including a field declaration for the unique (*disj*) position. The latter defines the position of the event in the process event chain. However, since it is possible to define relations

between atoms of complex signatures, too, the more intuitive implementation would be a *Linked List*. However, our performance tests showed that assigning just a unique position to each event is much faster than establishing predecessor and successor relations between atoms of signatures. Line 9 to 10 are examples of signature extensions which is comparable to inheritance in OOPLs. The signatures in line 9 and 10 are unique (keyword *one*) and denote the beginning and the completion of a process execution. Line 12 introduces the more interesting $TaskEvent$ for computing tuples consisting of a $TaskEvent$ atom, an integer atom which is the inherited or joined position as well as atoms for associated information like the executed $Task$ (cf. line 19) and the assigned organizational resource. The abstract $Task$ signature indicates that there should be at least one extending signature, which is part of the particular process model.

In order to distinguish different activity types like manual and automated tasks, the $TaskEvent$ signature is abstract. In line 15 $HumanTaskEvent$ is used to represent a manual task and it consequently extends the already discussed $TaskEvent$ signature. Both signatures have an appended fact which also could be formulated using an additional **fact** statement. However, this is only a matter of perspective, readability and personal preferences [16]. Line 13 ensures that a $TaskEvent$ encapsulates exactly one task and line 16 restricts the resources involved in the task execution to exactly one.

The lines 21-25 encode the functionality to specify data objects and write accesses to these data objects. We decided to extend a more general access type ($DataAccess$) in order to allow for extending the meta-model with different access types like, e.g., a read access.

However, running the solver would still produce undesired PCEs such as chains where the process start event has predecessors or the process end event has successors. Since, this cannot be avoided via structure definitions we make use of *facts* which are comparable to *invariants* in the *Object Constraint Language (OCL)*. Line 28-30 ensure that a process event chain starts with a $StartEvent$ (line 28) and ends with an $EndEvent$ (line 29 and 30) and consequently force all $TaskEvents$ to occur between. The third fact ensures that the position increment between two consecutive tasks is 1. The remaining three facts ensure that the solver only generates process elements that are "used" in at least one event (line 32-34) and prevents all events from containing information about organizational structures (line 35).

The first two utility functions calculate all $TaskEvent$s that involve the execution of a given task *in general* (line 38-40) or *before* (line 41-43) a given event. Line 44-46 define the function $roleOf$ which calculates all roles a particular resource has. The last function identifies the concrete $DataAccess$ signature for the given $DataObject$ and type.

In the following subsection the transformation of DPIL models to Alloy is described. It is based on the meta-model for process event chains introduced in the current subsection.

DPIL	Alloy
`task T`	`sig T extends Task{}`
`use identity I`	`lone sig I extends Identity{}`
`milestone event(T)`	`fact { ∀e: TaskEvent•#exist[T]=1 and #(T&e.assoEl)=0 →not(e.pos>exist[T].pos) }`
`sequence(T,U)`	`fact{ ∀e: TaskEvent•U in e.assoEl →#inBefore[e,T]>0 }`
`produces(T,d)`	`fact{ ∀e: TaskEvent•T in e.assoEl →dAccess[d,WriteAccess] in e.⋆assoEl }`
`consumes(T,d)`	`fact{ ∀e: TaskEvent•T in e.assoEl →#inBefore[e,dAccess[d,WriteAccess]] > 0 }`
`once(T)`	`fact{ lone e: TaskEvent•T in e.assoEl }`
`role(T,r)`	`fact{ ∀e: TaskEvent•e.task=A →r in roleOf(e.executor) }`
`binding(T,U)`	`fact{ ∀e,f: TaskEvent• T in e.assoEl and U in f.assoEl →#((e.assoEl & Identity) & f.assoEl) =1 }`

Table 2: Mapping: DPIL - Alloy

3.2 Transformation of DPIL models to Alloy

TODO: REWORK FROM HERE

The mapping of DPIL models to an Alloy model that conforms to our meta models from the previous paragraph involves two major steps: *(i)* Creating signatures for tasks, roles and identities that fulfill these roles and *(ii)* translating the DPIL rules to Alloy facts. Since, it would go beyond the scope to present an automatic translation, we mapped all DPIL elements necessary for our subsequent evaluation manually to Alloy (cf. Tab. 2). Tasks are modeled through the definition of a new signature that extends the existing `Task` signature from the meta model. The same is applicable to DPIL's identities but with the extension of the `Identity` signature instead. Tasks and identities are not shown in Fig. **??** due to their trivial semantics. DPIL milestones can be modeled as Alloy facts. DPIL rules are modeled as Alloy facts, too. They are specified in a declarative style through first selecting atoms that belong to certain signatures. Using the logical *implication* ($\rightarrow$) operator allows for specifying rule activation conditions (left part) and validity conditions (right part). In order to don't repeat ourselves too much, we make use of the functions contained in the process chain meta model, e.g. $evBefore$ and $roleOf$. Since there is currently no automatic transformation from DPIL to Alloy, we tested our manually translated constraints in isolation through the generation of a large set of examples (> 10000). Using the corresponding DPIL mining tool we successfully reconstructed the corresponding DPIL rule in all cases. For further information about the used Alloy syntax we would like to refer to the dedicated literature [16] as well as the vivid tutorial[2]. Within the next paragraph, we discuss, how the presented simulation approach can be utilized for the intended translation approach.

[2] available at: `http://alloy.mit.edu/alloy/tutorials/online/`

3.3 Simulation purposes: Process Mining Evaluation, history-awareness and hypotheses

3.4 Simulation Configuration

There are two major simulation parameters that are important for the whole translation approach *(CP3)*: *(i)* the *Number of simulated Traces (N)* and *(ii)* the *Maximum Trace Length (L)*. The provided rule of thumb for an appropriate parameter setting has been developed considering the number of different activities in the model without taking the structure into account. One motivation for the whole translation approach is that a business analyst is not necessarily familiar with a particular process modeling language. Thus, we based our formula on the stipulation to require as little language knowledge as possible. Since at least the number of different activities can be calculated automatically, we actually don't require any knowledge about the source language for an appropriate configuration of the simulation component.

However, the mentioned formula does not take into account additional trace variations though the organizational perspective. Hence, the formula must be extended. Hence, we base the extended formula on the idea that each possible selection of activities from all available activities T can be executed in random order and by an arbitrary selection of resources. The resulting formulae for the number of traces N, shown in Eq. 1, considers the number of different activities $|T|$ and an arbitrary involvement of resources O. It therefore considers the complete set of observable events [20] which is calculated via the Cartesian product.

$$N \geq \sum_{i \leq L}^{i=0} |T \times O|^i \qquad (1) \qquad\qquad L \geq 2 \cdot max(|T|, |O|) \qquad (2)$$

Consequently, the formula for the maximum trace length should be extended as shown in Eq. 2. The intuition for basing the maximum number of executed activities per trace is to ensure that all activities and all resources may co-occur. However, even this *lower bound* of required traces increases exponentially with increasing numbers of activities and resources. Since these formulae are only an initial rule of thumb, we do not prove them but provide more suitable settings in our evaluation section.

Simulation Configuration In 3.4 we discussed two essential parameters for the simulation part of our translation approach. With our Alloy-based simulation component it is not possible to configure the number of simulated traces directly since the Alloy Analyzer produces exactly one solution but allows for iterating over all possible solutions for the given maximum trace length L. The latter parameter, must be configured via the scope of the Alloy *run* command, which in our case is: `run` $<$ *exgen* $>$ `for` $<$ L $>$ `TaskEvent,` $<$ B $>$ `int`. The *exgen* parameter is the name of an empty predicate. The Analyzer only produces examples that fulfill this predicate - which is always the case since the predicate does not involve any condition. We also could encode all process rules in this predicate but since we already transformed them from DPIL to Alloy facts, this

would be redundant. Hence, the maximum number of executed activities per trace can be configured directly using the `for` $< L >$ `TaskEvent` part of the command. However, since we identify the position of an event in the process event chain via an index, we also have to provide the number of integer values to generate. This is done via the *bitwidth* parameter B of the run command. The Analyzer then generates integer values in the codomain of $\left[\frac{-2^B}{2}+1, \frac{2^B}{2}\right]$. Consecutively B can be calculated directly according to $B = \lceil \mathrm{ld}\, L \rceil$. To be able to translate a model in the opposite direction, we discuss the simulation of multi-perspective imperative models within the following subsection.

4 Conclusion and Future Work

The approach presented in this paper makes use of well-established techniques in process simulation and mining in order to translate a given business process model to a particular target language. Process simulation techniques are used to generate exemplary execution traces the source process model allows. This represents an easy-to-use alternative to conventional model-to-model transformation system without backtracking from target model elements to source model elements. Since plain control-flow models neglect dependencies between process participants as well as between process participants and activities we primarily focused on models that consider the behavioral *and* the organizational perspective. The presented principle is neither limited to languages nor to the imperative-declarative setting in general. Technically the applicability of our approach depends on the availability of corresponding simulation and mining technologies as well as an appropriate configuration. Since, a simulation technique for resource-aware declarative process models did not exist, yet, we introduced a limited draft based on the Alloy language. The simulation approach is not limited to the two considered process perspectives and it is planned to extend it in order to simulate multi-perspective declarative process models in general.

Acknowledgments. The authors would like to thank Prof. Westfechtel (University of Bayreuth) for providing access to literature about Alloy as well as Prof. Daniel Jackson (MIT) for tips regarding modeling in Alloy.

References

1. E. Verbeek, J. Buijs, B. van Dongen, and W. van der Aalst, "XES, xESame, and ProM 6," in *Information Systems Evolution*, pp. 60–75, 2011.
2. A.-W. Scheer and M. Nüttgens, *ARIS architecture and reference models for business process management*. Springer, 2000.
3. "Generating Event Logs through the Simulation of Declare Models the Simulation of Declare Models," *EOMAS*, pp. 20–36, 2015.
4. O. M. G. (OMG), "Business process model and notation (bpmn) version 2.0," tech. rep., jan 2011.

5. T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.
6. M. Pesic, H. Schonenberg, and W. van der Aalst, "DECLARE: Full Support for Loosely-Structured Processes," in *EDOC*, pp. 287–, Washington, DC, USA: IEEE Computer Society, 2007.
7. D. Fahland, D. Lübke, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal, "Declarative versus imperative process modeling languages: The issue of understandability," in *BPMDS*, pp. 353–366, 2009.
8. P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, and H. Reijers, "Imperative versus declarative process modeling languages: An empirical investigation," *BPM Workshops*, pp. 383–394, 2012.
9. F. M. Maggi, J. C. Bose, and W. van der Aalst, "A Knowledge-Based Integrated Approach for Discovering and Repairing Declare Maps," in *Advanced Information Systems Engineering*, pp. 433–448, 2013.
10. C. D. Ciccio and M. Mecella, "On the discovery of declarative control flows for artful processes," *ACM Transactions on Management Information Systems (TMIS)*, vol. 5, no. 4, p. 24, 2015.
11. M. Zeising, S. Schönig, and S. Jablonski, "Towards a Common Platform for the Support of Routine and Agile Business Processes," in *CollaborateCom*, 2014.
12. S. Schönig, C. Cabanillas, S. Jablonski, and J. Mendling, "Mining the organisational perspective in agile business processes," in *BPMDS*, pp. 37–52, 2015.
13. S. Schönig, A. Rogge-Solti, and J. Mendling, "SQL Queries for Declarative Process Mining on Event Logs of Relational Databases," *arXiv preprint arXiv:1507.03415*, 2015.
14. S. Jablonski, "MOBILE: A modular workflow model and architecture," in *Working Conference on Dynamic Modelling and Information Systems*, 1994.
15. W. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*, vol. 2. 2011.
16. D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.
17. W. M. P. van der Aalst, "Business Process Simulation Revisited," *Enterprise and Organizational Modeling and Simulation*, vol. 63, pp. 1–14, 2010.
18. R. Stewart, *Simulation: the practice of model development and use*. John Wiley & Sons, 2004.
19. C. Bussler, "Analysis of the organization modeling capability of workflow-management-systems," in *Proceedings of the PRIISM96 Conference*, pp. 438–455, 1996.
20. A. Burattin, A. Sperduti, and M. Veluscek, "Business models enhancement through discovery of roles," in *Computational Intelligence and Data Mining (CIDM), 2013 IEEE Symposium on*, pp. 103–110, IEEE, 2013.