

Simulation of Multi-perspective Declarative Process Models

Lars Ackermann^(✉), Stefan Schönig, and Stefan Jablonski

University of Bayreuth, Bayreuth, Germany

{lars.ackermann, stefan.schonig, stefan.jablonski}@uni-bayreuth.de

Abstract. Flexible business processes can often be represented more easily using a declarative process modeling language (DPML) rather than an imperative language. Process mining techniques can be used to automate the discovery of process models. One way to evaluate process mining techniques is to synthesize event logs from a source model via simulation techniques and to compare the discovered model with the source model. Though there are several declarative process mining techniques, there is a lack of simulation approaches. Process models also involve multiple aspects, like the flow of activities and resource assignment constraints. The simulation approach at hand automatically synthesizes event logs that conform to a given model specified in the multi-perspective, declarative language DPIL. Our technique translates DPIL constraints to a logic language called Alloy. A formula-analysis step is the actual log generation. We evaluate our technique with a concise example and describe an alternative configuration to simulate event logs based on an assumed partial execution as well as on properties that are intended to be checked. We complement the quality evaluation by a performance analysis.

Keywords: Simulation of business processes · Predictive analytics · Multi-perspective process mining

1 Introduction

Business process simulation supports those phases of the business process management lifecycle that aim at the analysis and improvement of processes [1]. New versions of processes are simulated in order to determine an optimal improvement. Logs produced by simulating processes are analyzed in order to predict effectiveness or efficiency of upcoming versions of processes. Besides analysis, another purpose of process simulation is learning about the meaning of a process. By simulating processes, modelers and users can learn to understand their behavior based on selected log contents [2]. From cognitive science we learn that studying and observing “good examples” of artifacts, here processes, develop their comprehension [3]. A third purpose of simulation is its support for testing process mining techniques [4]. Di Ciccio et al. [5] propose to use simulation in order to generate process logs that are used to test and improve process mining algorithms. It becomes obvious that simulation plays an important role in the lifecycle of business process management.

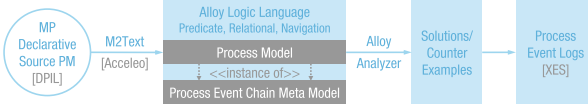


Fig. 1. Concept of multi-perspective declarative process model simulation

Most of the available simulation techniques are tailored towards imperative languages such as *BPMN*, e.g. [6,7]. Over the last years, *declarative* process modeling languages (DPMLs) [8–10] and declarative process discovery techniques gained more and more attraction [4, 11, 12]. Imperative languages model the underlying process *explicitly* using flow-oriented representations. In contrast, declarative languages assume process executions which are restricted by constraints. Due to this semantic gap, the transformation of, especially multi-perspective, declarative process models to an appropriate imperative representation is still vague [13]. Consequently, simulation techniques for imperative models are not suitable for declarative models [5] which leads to a lack of simulation tools for the latter. The approach presented in [5] is the only representative that is able to generate traces based on rules that restrict the temporal ordering and the existence of activities. The simulator and the underlying modeling language consider only control-flow constraints but no other process perspectives including organizational and data-oriented aspects [10]. To the best of our knowledge a multi-perspective declarative process simulation technique is not available.

The approach visualised in Fig. 1 fills this research gap with a simulation technique for *multi-perspective* process models that is based on *Declarative Process Intermediate Language (DPIL)* [10]. We based our simulation technique on a transformation of DPIL rules to a logic language called *Alloy* [14]. An important advantage of using Alloy, in contrast to simulation tools designed for imperative models, is that DPIL rules and Alloy logic expressions have a direct correspondence. Alloy ships with an analyzer that is able to *exhaustively* produce unique examples and counter examples for a given Alloy model. It is possible to produce logs with desired characteristics like size, maximum trace length, trace contents or relative to a partial process execution trace.

This paper is structured as follows: Sect. 2 introduces DPIL. The discussion of the contribution in Sect. 3 is based on a brief description of Alloy (cf. Sect. 2.2). The evaluation is described in Sect. 4 and the paper is concluded in Sect. 5.

2 Background

In this section we introduce the foundations of our approach, i.e., declarative process modelling and DPIL as well as Alloy.

2.1 Multi-perspective Declarative Process Modeling with DPIL

Research has shown that DPMLs are able to cope with a high degree of flexibility [8]. The basic idea is that, without modeling anything, “everything is

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

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 $<p$)	Task a can be started once only
Consumes(c,i)	event(of c at $:t$) implies write(of i at $<t$)	Task c can not be started before a data object i has a value
Produces(p,o)	event(of p : t) implies write(of o at $<t$)	Task p can not be completed before a value for the data object o is present
Role(a,r)	event(of a by $:id$) implies relation(subject id predicate <i>hasRole</i> object r)	Task a must be performed by a process participant in role r
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

allowed". To reduce this flexibility a declarative process model contains constraints which form a forbidden region for process execution paths. 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. Another perspective is the data-oriented one which deals with restrictions regarding the data flow. The *Declarative Process Intermediate Language (DPIL)* [10] is a multi-perspective declarative process modelling language, i.e., it allows for representing several business process perspectives, namely, control flow, data and especially resources in one model. Comparable languages are data-aware Declare [15] and ConDec-R [16]. In contrast to DPIL, the former only allows for formulating control-flow and data-constraints and the latter provides support only for control-flow restrictions and resource allocation constraints. The expressiveness of DPIL and its suitability for business process modelling have been evaluated [10] with respect to the well-known Workflow Patterns and in industry projects, e.g. the *Competence Center for Practical Process Management*. Although we selected DPIL as our example language, the principle is also applicable to other rule-based process modeling languages.

DPIL provides a *macro* based textual notation to define reusable rules, shown exemplarily in Table 1. We explain all macros using the example process model of Fig. 2 which shows a simple process for trip management in DPIL. The process model states, for instance, that it is mandatory for all applicants to produce the application document for a business trip before it can be approved (*produces* and *consumes*). Means of transport and accommodations can only be booked after the application has been approved (*sequence*). Every task except booking accommodations and means of transport can be performed at most once (*once*).

```

use group Administration

process BusinessTrip {
  task Apply for Trip
  task Approve application
  task Book means of transport
  task Book accommodation
  task Collect tickets
  task Archive documents
  document Application
  document TicketCollection

  ensure produces(Apply for Trip, Application)
  ensure produces(Collect tickets, TicketCollection)
  ensure consumes(Approve application, Application)
  ensure consumes(Archive documents, Application)
  ensure consumes(Archive documents, TicketCollection)

  ensure sequence(Approve application, Book means of transport)
  ensure sequence(Approve application, Book accommodation)
  ensure sequence(Book means of transport, Collect tickets)
  ensure sequence(Book accommodation, Collect tickets)
  ensure once(Apply for Trip)
  ensure once(Approve application)
  ensure once(Collect tickets)
  ensure once(Archive documents)

  ensure role(Approve Application, Administration)
  ensure binding(Book means of transport, Apply for Trip)
  ensure binding(Book accommodation, Apply for Trip)
  ensure binding(Collect tickets, Apply for Trip)
  ensure binding(Archive documents, Apply for Trip)

  milestone "Done": event(of Collect tickets) and event(of Archive documents)
}

```

Fig. 2. Process for trip management modeled with DPIL

The latter can be executed multiple times in order to allow, e.g., for flights with stopover and multiple accommodations per trip. The task *Approve application* must be performed by a resource with the role *Administration*. Additionally it is required that the same person – here the applicant – books the flight and the accommodation (*binding*). In the described setting there is no secretary which is why the applicant is also responsible for collecting the tickets and for archiving the collected documents. A process instance is finished as soon as the tickets are collected and all documents are archived (*milestone*).

2.2 Alloy in a Nutshell

Alloy is a declarative language for building models that describe structures with respect to desired restrictions. We first provide a concise and pragmatic description of Alloy's language features: A signature (*sig*) is similar to a class in object-oriented programming languages (OOPLs). It can be abstract and quantified. A *fact* is comparable to *invariants* in the *Object Constraint Language (OCL)* [17] and allows for specifying non-structural constraints. A function (*fun*) is a parameterizable snippet of re-usable code, that has a return type and performs computations based on the given parameter values. A predicate (*pred*) is comparable

to a function but with the limitation that its return type is always a boolean expression. An additional major difference is that Alloy is able to *run* a predicate, which means that the analyzer tries to find models for which that predicate holds. An assertion (*assert*) can be used in combination with *check* commands to test model properties. The body of facts and assertion share the same syntax but in contrast to the former, the analyzer tries to find counter examples for a particular assertion. For further information about the general Alloy syntax we would like to refer to the dedicated literature [14].

3 Simulation of DPIL Models with Alloy

Due to Alloy’s declarative nature, it can be used to represent a declarative process model. The correspondence between DPIL and Alloy as well as a mapping are described within this section, starting with a concise characterization.

3.1 Requirements and Functional Characteristics

Process simulation is used for the analysis of properties of business processes [1]. Our approach supports process analysis through event log generation. We identified the following requirements based on the introductory simulation purposes:

- *Distinctness*. Distinctness means to avoid redundant traces. This feature keeps the set of examples as small as possible. Without this feature a log can grow enormously without enhancing information content; its growth then worsen its performance and clarity.
- *Exhaustiveness*. This feature guarantees that all possible process execution paths of a defined maximum length are considered.
- *Determinism*. Determinism says that parts of the log can be replicated according to user defined settings. This is needed to specifically weight alternative execution paths.
- *Multi-perspectivity*. Processes are constituted by multiple perspectives [10]. These perspectives must be identifiable in a process log.
- *Context-awareness*. This property allows to analyze traces taking into account particular process states. Such a process state might depict a certain (partial) execution path; the log then should be analyzed whether there are processes coinciding with that execution path. For instance, if such an execution path depicts the beginning of a process trace, this analysis ascertains whether this process will eventually terminate (i.e. a process trace must be found that shows this prefix and reaches an end state).
- *Reversibility*. It can be useful to generate traces that explicitly violate process specifications (counter examples). From cognitive science we adopt that counter examples are good for gaining understanding (here: of processes) [18].

By basing the simulation on Alloy [14] the first two properties, distinctness and exhaustiveness, are guaranteed. As a consequence, determinism is incidentally achieved, too. The remaining two characteristics are explained further in Sect. 3.4.

3.2 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. In Alloy we defined our meta model for traces in form of *process event chains (PECs)* in three modules. Two of them are shown in Listings 1.1 and 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.

Listing 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 where process resources [20] may be involved in based on a *subject-predicate-object (spo)* notation. An example would be: *John (s) hasRole (p) Admin (o)*. In our corresponding Alloy-based process model we need four additional signatures in order to represent an instance of this relation – one for *Relation* itself and one for each of the contained fields.

```

1 module orgmetamodel
2 open processEventChain_commons
3
4 abstract sig Relation {s: one Element, p: one RelationType, o: one Element}
5 abstract sig Element extends AssociatedElement {}
6 abstract sig Identity extends Element{}
7 abstract sig Group{} extends Element{}
8 abstract sig RelationType{}

```

Listing 1.1. Organizational Meta-model

The structure of PECs was mainly motivated by the log structures discussed in [21] as well as related literature and is described in Listing 1.2. After defining the module name we make the two previously described modules available (line 2 and 3). The lines 5–17 describe the structural and the remaining lines describe the non-structural properties of a PEC.

PEvent is an abstract “class” for a general discrete event, including a field declaration for the unique (*disj*) position. The latter defines the position of the event in the PEC. Alternatively, a more intuitive implementation would be a *Linked List*. However, our performance tests showed that the proposed variant is much faster. The signatures in line 7 and 8 are unique (keyword *one*) and denote the beginning and the completion event of a process execution. Line 9 introduces the more interesting *TaskEvent* denoting an activity execution and comprising an integer which is the inherited position as well as associated information like the executed *Task* (cf. line 13) and the assigned organizational resource. The *Task* signature is abstract and is extended in the actual Alloy process model in order to represent concrete tasks (cf. Table 2). In order to

```

module processEventChain_noLifecycle_multiperspect_IntBased_new
2 open processEventChain_commons
open orgmetamodel
4
// Signatures: Process Chain Element Structure
6 abstract sig PEvent { pos: disj Int }
one sig StartEvent extends PEvent{}{}
8 one sig EndEvent extends PEvent{}{}
abstract sig TaskEvent extends PEvent { assoEl: some AssociatedElement }{
10     #(Task & assoEl) = 1 }
sig HumanTaskEvent extends TaskEvent{}{
12     #(Identity & assoEl) = 1 }
abstract sig Task extends AssociatedElement{}
14 abstract sig DataObject {}
abstract sig DataAccess extends AssociatedElement{ data: one DataObject }
16 abstract sig WriteAccess extends DataAccess{}

18 // Facts: Additional non-structural constraints
fact {  $\forall$  intVal: Int • intVal  $\geq$  StartEvent.pos }
20 fact {  $\forall$  e: (PEvent - StartEvent - EndEvent) •
    e.pos < (StartEvent.pos + #TaskEvent + 1) }
22 fact { EndEvent.pos  $\leq$  (StartEvent.pos + #TaskEvent + 1) }
fact {  $\forall$  assoEls: (AssociatedElement - Group) •
24     assoEls in TaskEvent.assoEl }
fact {  $\forall$  do: DataObject • do in DataAccess.data }
26 fact {  $\forall$  te: TaskEvent • #(te.assoEl & Group) = 0 }

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

```

Listing 1.2. Process Event Chain Meta Model

distinguish between different activity types like manual and automated tasks, the *TaskEvent* signature is abstract, too. In line 11 *HumanTaskEvent* is used to represent a manual task and it consequently extends the *TaskEvent* signature. Both signatures have an appended fact which also could be formulated using an additional *fact* statement which is only a matter of personal preferences [14]. The appended facts ensure that a *TaskEvent* encapsulates exactly one task (line 10) and one executing resource (line 12). The lines 14–16 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 read accesses.

The lines 19–21 ensure that a process event chain starts with a *StartEvent* (line 19) and ends with an *EndEvent* (lines 20–21) and consequently force all *TaskEvents* to occur in 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 (lines 23–25) and prevents all events from containing information about organizational structures (line 26), since the organizational structures can be defined using the organizational meta-model shown in Listing 1.1.

The first two utility functions collect all *TaskEvents* that involve the overall execution of a given task (lines 29–30) or *before* (lines 31–32) a given event. The function *roleOf* calculates all roles a particular resource has. The last function identifies the concrete *DataAccess* signature for the given *DataObject* and type.

3.3 Transformation of DPIL Models to Alloy

Based on the process event chain meta-model presented above, we now discuss how to transform a DPIL model into an Alloy model that contains all restrictions for *valid* process event chains. This involves two major steps: (i) Creating signatures for tasks, roles and identities that fulfill these roles, data objects and access objects and (ii) translating the DPIL rules to Alloy facts (cf. Table 2).

Table 2. Mapping: DPIL - Alloy

DPIL	Alloy
task T	sig T extends Task{}
use group G	sig G extends Group{ one sig HasRole extends RelationType { } abstract sig IsG extends Relation { } { o =G \wedge p =HasRole }
document d	sig d extends DataObject{ sig Write_d extends DataAccess{ { data =d }
sequence (T,U)	fact { $\forall e$: TaskEvent • U in e.assoEl \rightarrow #inBefore[e,T] > 0 }
produces (T,d)	fact { $\forall e$: TaskEvent • T in e.assoEl \rightarrow dAccess[d,WriteAccess] in e.*assoEl }
consumes (T,d)	fact { $\forall e$: TaskEvent • T in e.assoEl \rightarrow #inBefore[e,dAccess[d,WriteAccess]] > 0 }
once (T)	fact { lone e: TaskEvent • T in e.assoEl }
role (T,r)	fact { $\forall e$: TaskEvent • e.task=A \rightarrow r in roleOf(e.executor) }
binding (T,U)	fact { $\forall e, f$: TaskEvent • T in e.assoEl and U in f.assoEl \rightarrow #((e.assoEl & Identity) & f.assoEl) = 1 }
milestone event (T)	fact { $\forall e$: TaskEvent • #exist [T]=1 and # (T&e.assoEl)=0 \rightarrow not (e.pos > exist [T].pos) }

Tasks are modeled by extending the existing Task signature from the meta model. In a similar way DPIL’s *use group* and *documents* are mapped by extending the Group and DataObject signatures, respectively. In order to type data accesses, we additionally extend the *DataAccess* signature. Additionally a new *Relation* signature is created to be able to easily assign a role to the desired resources (*Identity*). Using this mapping it is only possible to represent flat resource-role associations. However, based on the generic organizational meta-model shown in Listing 1.1 it is possible to model hierarchical structures, too.

DPIL rules are modeled as Alloy *facts*. Alloy rules are declarative and first select atoms belonging to particular signatures the rule shall be applied to. Using the logical *implication* (\rightarrow) operator one can specify rule activation (left part) and validity conditions (right part). In order keep the rules concise, we make use

of the functions contained in the process event chain meta-model like *inBefore*. The current simple milestone transformation considers milestones that can be reached when executing particular activities. Since *facts* are connected via conjunction we can generate one fact per activity execution that is observed by a milestone rule.

3.4 Simulation Configuration

There are two simulation parameters that are required in most cases [5]: (i) The number of simulated traces (N) and (ii) the maximum trace length (L). Restricting the log size in terms of the number of traces is necessary to be able to provide a reproducible setting for trace generation. The number of events per trace should be restricted because of potential infinite activity loops. Furthermore, the aspect of reproducibility is also influenced by the trace length. Beside these essential simulation boundaries additional parameters may be useful, dependent on the simulation purpose. Though it is impossible to guess the particular simulation purpose (cf. Sect. 1), this section describes three different configurations: (i) Trace generation, (ii) context-aware simulation and (iii) property testing.

Using Alloy trace generation can be implemented by introducing an empty predicate (*sim*) and configuring a *run* command. This can be done according to the following template: **run** *sim* **for** [L] *TaskEvent*, [B] *Int*. The introduced *length* parameter L can be configured directly through a scope restriction for *TaskEvents*. 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 . The Analyzer then generates integer values in the codomain of $\left[-\frac{2^B}{2} + 1, \frac{2^B}{2}\right]$. Hence, B can be calculated directly according to $B = \lceil \text{ld } L \rceil$. Via collecting all unique results produced by the Alloy analyzer the desired amount of traces can be obtained.

Here, a *context-aware* simulation means that the simulation is not started at the beginning of a particular process but “somewhere between” the start and the end of the process. An example application is to check the satisfiability assuming a particular process state and to generate all traces that remain. This can be implemented by adding a *fact* for each assumed event that already happened and assigning a fixed position as well as *AssociatedElements* to an event at this position. The position can be calculated generically based on the position of the *StartEvent*. The simulation can be started using a *run* command, too.

A *hypothesis* is an assumption regarding structure and contents of a trace. In order to check hypotheses they have to be transformed into *predicates*. A predicate can be checked in an *assertion*. Instead of using a *run* command the *check* command has to be used but the parameters are the same. Running the analyzer results either in counter examples proving that a hypothesis is wrong or does not provide any result and, thus, corroborates a hypothesis. With this mode selected properties of the source model can be tested.

4 Implementation and Evaluation

In order to evaluate the simulation approach efficiently, we implemented a *model-to-text* transformation using *Acceleo*¹ which automatically translates DPIL models into Alloy. *Acceleo* is an implementation of the *MOF Model to Text Transformation Language (MOFM2T)*² defined by the OMG. The transformation is currently based on the macros discussed in the paper at hand. The generated Alloy file is then used in our simulator implementation³ to generate traces of a configurable length and amount and serializes them in the *eXtensible Event Stream (XES)* [22] standard format. In order to evaluate the correctness of the generated traces regarding the source process model we make use of the same evaluation principle as in [5]. This means that we use a previously evaluated process mining technology and try to reproduce the original process model. For the paper at hand we utilized *DPILMiner* [4]. As evaluation example we used the DPIL process model shown in Fig. 2. We configured *DPILMiner* with the same set of rule templates like the simulation approach. After applying transitive reduction techniques on the extracted model, *DPILMiner* reproduced exactly the source model. Additionally, we performed property tests for all generated facts which is comparable to unit testing. These property tests have been implemented using *assertions* and the *check* command. Another aspect of the evaluation is the performance of the proposed simulation technique. Since the simulation time increases with higher parameterizations for the number of traces (N) and their maximum lengths (L), we have performed several simulations of the continuous process model example with different configurations and results shown in Table 3.

The performance analysis shows that the computation is mainly influenced by the trace length. Furthermore, as a minor detail, we have no increase of computation time between the second and the third configuration (the time measurements in parentheses). The reason was that with a maximum trace length of 10 there are less than 100 different process event chains. For the performance analysis, we used a Dell Latitude E6430 (Core i7-3720QM with 8×2.6 GHz, 16 GB memory, SSD drive and Windows 8 64 Bit). The simulator is implemented in Java and we used a 64-Bit JVM with a maximum memory allocation pool of 4096M. We decided to present the performance analysis without a comparison to the technique discussed in [5] because there are large functional differences. First, the approach presented in the paper at hand considers multiple perspectives, which is not possible with the technique proposed in [5]. Secondly, our approach guaranties to simulate *all unique* traces of a defined maximum length. Additionally our simulation technique can be used in three different modes (cf. Sect. 3.4). These major functional differences result in an increase of computation time and in a significant decrease in terms of scalability. Thus, we can say that the approach presented in [5] should be used if you need event logs with longer traces that reflect the plain control flow. If the particular application involves multiple

¹ Download: <http://www.eclipse.org/acceleo>, last access: July 22, 2016.

² Standard: <http://www.omg.org/spec/MOFM2T/1.0/>, last access: July 22, 2016.

³ Screenshot and Download (incl. example data): <http://mps.kppq.de>.

Table 3. Performance analysis

L	N	Time in s	L	N	Time in s
10	10	1.9	50	10	364.9
10	100	(2.4)	50	100	389.9
10	1000	(2.4)	50	1000	555.8
20	10	17.4	60	10	627.3
20	100	21.3	60	100	649.5
20	1000	52.8	60	1000	871.5
30	10	65.8	70	10	1167.1
30	100	71.8	70	100	1271.5
30	1000	122.3	70	1000	1697.0
40	10	159.0	80	10	2038.8
40	100	180.0	80	100	2194.3
40	1000	300.0	80	1000	2733.5

perspectives, and either the trace length is rather low or the computation time is not a main concern we suggest to use the presented technique.

5 Conclusion and Future Work

The paper at hand describes a process simulation technique which can be used to generate exemplary execution traces for a given process model in order to support business process management. There is only one comparable approach and this considers only plain control-flow models. Our proposed simulation approach primarily focuses on models that consider the behavioral, the organizational, *and* the data-oriented perspective. Additionally to the generation of exemplary traces, the simulation can be used in two additional modes, i.e. *(i)* context-aware simulation and *(ii)* property testing. Both modes can be used for targeted process analysis or gaining a deeper general understanding of the underlying process. A generic meta-model for process event chains and an independent logic framework called Alloy opens the opportunity for extensions. An open issue is the rather low simulation performance and scalability in the case of longer process event chains. Similar to general purpose programming languages, the same functionality can be developed more or less efficiently, dependent on the programming style. Consequently, there is a huge potential for performance optimization, e.g. the order of set joins which is a known issue in databases. Hence, we are currently planning a major evaluation study in order to get a better idea of the driving factors for scalability. Another limitation is the small set of supported rule templates (macros). In order to check Alloy's applicability we formed the set as heterogeneous as possible. Thus, extending this initial set of macros should be rather straightforward. The presented technique focuses on trace generation

rather than process performance analysis. Conventional simulation tools emulate variability concerning activity durations and human decisions based on probability distributions which cannot be modeled using Alloy. Hence, we are currently developing a post-processing step which is able to compensate this limitation.

Acknowledgments. The authors would like to thank Prof. Westfechtel, Felix Schwägerl (University of Bayreuth) and Prof. Daniel Jackson (MIT) for providing tips and literature about modeling and analysis with Alloy.

References

1. van der Aalst, W.M.P.: Business process simulation revisited. *Enterp. Organ. Model. Simul.* **63**, 1–14 (2010)
2. Frank, U.: Multi-perspective enterprise modeling (memo) conceptual framework and modeling languages. In: HICSS, pp. 1258–1267 (2002)
3. Brown, A.L., Kane, M.J.: Preschool children can learn to transfer: learning to learn and learning from example. *Cogn. Psychol.* **20**, 493–523 (1988)
4. Schönig, S., Cabanillas, C., Jablonski, S., Mendling, J.: Mining the organisational perspective in agile business processes. In: *BPMDs*, pp. 37–52 (2015)
5. Di Ciccio, C., Bernardi, M.L., Cimitile, M., Maggi, F.M.: Generating event logs through the simulation of declare models. In: Barjis, J., Pergl, R., Babkin, E. (eds.) *EOMAS 2015. LNBIP*, vol. 231, pp. 20–36. Springer, Cham (2015). doi:[10.1007/978-3-319-24626-0_2](https://doi.org/10.1007/978-3-319-24626-0_2)
6. De Medeiros, A.A., Günther, C.W.: Process mining: using cpn tools to create test logs for mining algorithms. In: *Proceedings of CPN*, vol. 576 (2005)
7. Burattin, A., Sperduti, A.: PLG: a framework for the generation of business process models and their execution logs. In: Muehlen, M., Su, J. (eds.) *BPM 2010. LNBIP*, vol. 66, pp. 214–219. Springer, Heidelberg (2011). doi:[10.1007/978-3-642-20511-8_20](https://doi.org/10.1007/978-3-642-20511-8_20)
8. 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.) *BPMDs/EMMSAD -2009. LNBIP*, vol. 29, pp. 353–366. Springer, Heidelberg (2009). doi:[10.1007/978-3-642-01862-6_29](https://doi.org/10.1007/978-3-642-01862-6_29)
9. Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., Reijers, H.A.: Imperative versus declarative process modeling languages: an empirical investigation. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) *BPM 2011. LNBIP*, vol. 99, pp. 383–394. Springer, Heidelberg (2012). doi:[10.1007/978-3-642-28108-2_37](https://doi.org/10.1007/978-3-642-28108-2_37)
10. Zeising, M., Schönig, S., Jablonski, S.: Towards a common platform for the support of routine and agile business processes. In: *CollaborateCom* (2014)
11. Maggi, F.M., Bose, R.P.J.C., Aalst, W.M.P.: A knowledge-based integrated approach for discovering and repairing declare maps. In: Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) *CAiSE 2013. LNCS*, vol. 7908, pp. 433–448. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-38709-8_28](https://doi.org/10.1007/978-3-642-38709-8_28)
12. Schönig, S., Rogge-Solti, A., Cabanillas, C., Jablonski, S., Mendling, J.: Efficient and customisable declarative process mining with SQL. In: Nurcan, S., Soffer, P., Bajec, M., Eder, J. (eds.) *CAiSE 2016. LNCS*, vol. 9694, pp. 290–305. Springer, Cham (2016). doi:[10.1007/978-3-319-39696-5_18](https://doi.org/10.1007/978-3-319-39696-5_18)

13. Ackermann, L., Schöning, S., Jablonski, S.: Towards simulation- and mining-based translation of resource-aware process models. In: Proceedings of ReMa (2016)
14. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2012)
15. Montali, M., Chesani, F., Mello, P., Maggi, F.M.: Towards data-aware constraints in declare. In: Proceedings of the 28th SAC, pp. 1391–1396. ACM (2013)
16. Barba, I., Weber, B., Del Valle, C., Jiménez-Ramírez, A.: User recommendations for the optimized execution of business processes. *DKE* **86**, 61–84 (2013)
17. Warmer, J.B., Kleppe, A.G.: The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley OTS). Addison-Wesley Professional, Boston (1998)
18. Zazkis, R., Chernoff, E.J.: What makes a counterexample exemplary? *Educ. Stud. Math.* **68**(3), 195–208 (2008)
19. Bussler, C.: Analysis of the organization modeling capability of workflow-management-systems. In: PRIISM 1996 Conference Proceedings, pp. 438–455 (1996)
20. Object Management Group (OMG): Business process model and notation (bpmn) version 2.0. Technical report, January 2011
21. van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes, vol. 2. Springer, New York (2011)
22. Verbeek, H.M.W., Buijs, J.C.A.M., Dongen, B.F., Aalst, W.M.P.: XES, XESame, and ProM 6. In: Soffer, P., Proper, E. (eds.) CAiSE Forum 2010. LNBIP, vol. 72, pp. 60–75. Springer, Heidelberg (2011). doi:[10.1007/978-3-642-17722-4_5](https://doi.org/10.1007/978-3-642-17722-4_5)