Simulation of Multi-Perspective Declarative Process Models

Lars Ackermann, Stefan Schönig and Stefan Jablonski

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

Abstract Flexible business processes can often be modelled more easily using a declarative process modeling language (DPML) rather than an imperative language. Process mining aims at automating 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 comprises the log generation. We evaluate our technique with a concise example and describe how it can be configured to alternatively 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, 2]. 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 [3]. From cognitive science we learn that studying and observing “good examples” of artifacts, here processes, develop their comprehension [4]. A third purpose of simulation is its support for testing process mining techniques [5]. Di Ciccio et al. [6] 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.

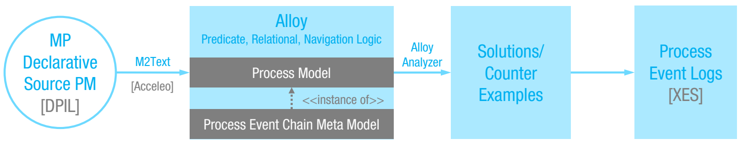

Fig. 1: Concept of Multi-Perspective Declarative Process Model Simulation

Most of the available simulation techniques are tailored towards imperative languages [6] such as *BPMN*. Over the last years, *declarative* process modeling languages (DPMLs) and declarative process discovery techniques gained more and more attraction [5, 7–13]. 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, simulation techniques for imperative models are not suitable for declarative models [6]. Consequently, there is a lack of simulation tools for declarative models. The approach presented in [6] 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 [12]. To the best of our knowledge a multi-perspective declarative process simulation technique is not available.

We fill this research gap by an approach visualised in Fig. 1. It is designed to complement the control-flow-based simulation with a technique that is based on the multi-perspective *Declarative Process Intermediate Language (DPIL)* [12] and is able to simulate *multi-perspective* process models. We based our simulation technique on a transformation of DPIL rules to a logic language called *Alloy* [14] which was originally used for describing the structure of software systems. 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 configure the simulator in order 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: Sec. 2 provides a brief introduction into the declarative modeling paradigm and DPIL. In order to be able to follow the contribution discussed in Sec. 3 we also provide a brief description of Alloy. The evaluation is described in Sec. 4 and the paper is concluded in Sec. 5.

2 Background

In this section we introduce the foundations of our approach, i.e., declarative process modelling and DPIL, 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 [8]. The basic idea is that, without modeling anything, "everything is

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.

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

allowed". To restrict this maximum flexibility, DPMLs allow for formulating 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 data-oriented and deals with restrictions regarding the data flow. The *Declarative Process Intermediate Language (DPIL)* [12] 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 [12] with respect to the well-known Workflow Patterns and in industry projects, e.g. the *Competence Center for Practical Process Management.* DPIL provides a textual notation based on the use of *macros* to define reusable rules, shown exemplarily in Tab. 1. Instead of explaining the macros in isolation, we discuss them 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*). 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*).

```
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

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)* [15] 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 assertions (*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 in order to avoid an expensive observation of process executions [1]. Our approach provides process analysis support through event log generation. We identified the following requirements based on the introductory simulation purpose:

– *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 [12]. 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) [16].

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 Sec. 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

```
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

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 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.

Lst. 1.1 is the Alloy implementation of the well known organizational meta-model introduced in [17]. 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 [18] 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 an instance of this relation – one for *Relation* itself and one for each of the contained fields.

The structure of PECs was mainly motivated by the log structures discussed in [19] as well as related literature and is described in Lst. 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.

From the perspective of object-oriented programming *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. 2). In order to distinguish between different activity types like manual and automated tasks, the *TaskEvent* signature is abstract, too.

```
   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: Additonal non-structural constraints
   fact { ∀ intVal: Int • intVal ≥ StartEvent.pos }
20 fact { ∀ e: (PEvent - StartEvent - EndEvent) •
           e.pos < (StartEvent.pos + #TaskEvent + 1) }
22 fact { EndEvent.pos ≤ (StartEvent.pos + #TaskEvent + 1) }
   fact { ∀ assoEls: (AssociatedElement - Group) •
24         assoEls in TaskEvent.assoEl }
   fact { ∀ do: DataObject • do in DataAccess.data }
26 fact { ∀ 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.subject=id and r.object 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

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).

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 {} {`
	    `object =G`
	    `predicate =HasRole      }`

`document d`	**sig** `d` **extends** `DataObject{}` **sig** `Write_d` **extends** `DataAccess{}{ data =d }`
`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 }`
`milestone event(T)`	**fact** `{ ∀e: TaskEvent●#exist[T]=1` **and** `#(T&e.assoEl)=0` `→`**not**`(e.pos>exist[T].pos) }`

Tab. 2: Mapping: DPIL - Alloy

The first two utility functions collect all $TaskEvent$s 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

After providing a meta-model for process event chains, 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. 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 *usegroup* but with the extension of the `Group` signature instead. Additionally a new *Relation* signature is created in order to be able to easily assign a role to the desired resources ($Identity$ in our meta-model). Using this mapping it is only possible to represent flat organizational structures like resource-role associations. However, based on the generic organizational meta-model shown in Lst. 1.1 it would be possible to model hierarchical structures, too. A DPIL document is mapped to a new signature extending the existing *DataObject* signature. In order to type data accesses, we additionally extend the *DataAccess* signature.

DPIL rules are modeled as Alloy $fact$s. They are specified in a declarative style through first selecting atoms that belong to particular signatures. Using the logical *implication* ($\rightarrow$) operator allows for specifying rule activation conditions (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, e.g. $inBefore$ and $roleOf$. The current simplified milestone transformation considers milestones that can be reached through the execution of particular activities. Since $fact$s 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 [6]: *(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 due to the reason that process models might allow for executing an activity arbitrarily often. This means that the simulation would not necessarily terminate in all cases. 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 we are not able to determine the purpose for simulating a DPIL model, this section describes configurations for three simulation types: *(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 $TaskEvent$s. 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 $B = \lceil \mathrm{ld}\, L \rceil$. 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 *pred*icates. 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 hypotheses are wrong or does not provide any result and, thus, corroborates a hypotheses. 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] in order to automatically translate 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 implementation[3] to generate traces of a configurable length and amount. In order to use the logs in applications that are built upon a particular log standard, the simulator exports the traces in the *eXtensible Event Stream (XES)* [20] 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 [6]. 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 the *DPILMiner* [5]. As evaluation example we used the DPIL process model shown in Fig. 2. We configured the DPILMiner with the same set of rule templates like the simulation approach. After applying transitive reduction techniques on the extracted model, the DPILMiner reproduced exactly the source model. Additionally, we performed one property test for each Alloy representation of a DPIL rule which is comparable to unit testing. These property tests have been implemented based on *assert*ions and the *check* command. Another aspect of the evaluation is the performance of the proposed simulation technique. Since the simulation time increases with higher paramterizations 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 Tab. 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 then 100 different process event chains. For the performance analysis, we used a Dell Latitude E6430 (Core i7-3720QM with 8 × 2.6GHz, 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 [6] 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 [6]. 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. Sec. 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

[1] Download: `http://www.eclipse.org/acceleo`, last access: June 6, 2016

[2] Standard: `http://www.omg.org/spec/MOFM2T/1.0/`, last access: June 6, 2016

[3] Screenshot and Download: `http://mps.kppq.de`

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

Tab. 3: Performance Analysis

the approach presented in [6] should be used if you need event logs with longer traces that reflect the plain control flow. If the particular application involves multiple 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

In the paper at hand, we discussed 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.

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. W. M. P. van der Aalst, "Business Process Simulation Revisited," *Enterprise and Organizational Modeling and Simulation*, vol. 63, pp. 1–14, 2010.
2. M. Laguna and J. Marklund, *Business process modeling, simulation and design.* CRC Press, 2013.
3. U. Frank, "Multi-perspective enterprise modeling (memo) conceptual framework and modeling languages," in *HICSS*, pp. 1258–1267, 2002.
4. A. L. Brown and M. J. Kane, "Preschool children can learn to transfer: Learning to learn and learning from example," *Cogn. Psychology*, vol. 20, pp. 493–523, 1988.
5. S. Schönig, C. Cabanillas, S. Jablonski, and J. Mendling, "Mining the organisational perspective in agile business processes," in *BPMDS*, pp. 37–52, 2015.
6. C. Di Ciccio, M. L. Bernardi, M. Cimitile, and F. M. Maggi, "Generating event logs through the simulation of declare models," in *EOMAS*, pp. 20–36, 2015.
7. M. Pesic, H. Schonenberg, and W. van der Aalst, "DECLARE: Full Support for Loosely-Structured Processes," in *EDOC*, 2007.
8. 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.
9. 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.
10. 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.
11. C. D. Ciccio and M. Mecella, "On the discovery of declarative control flows for artful processes," *ACM TMIS*, vol. 5, no. 4, p. 24, 2015.
12. M. Zeising, S. Schönig, and S. Jablonski, "Towards a Common Platform for the Support of Routine and Agile Business Processes," in *CollaborateCom*, 2014.
13. S. Schönig, A. Rogge-Solti, C. Cabanillas, S. Jablonski, and J. Mendling, "Efficient and Customisable Declarative Process Mining with SQL," in *CAiSE*, 2016.
14. D. Jackson, *Software Abstractions: logic, language, and analysis.* MIT press, 2012.
15. J. B. Warmer and A. G. Kleppe, *The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley OTS).* Addison-Wesley Professional, 1998.
16. R. Zazkis and E. J. Chernoff, "What makes a counterexample exemplary?," *Educational Studies in Mathematics*, vol. 68, no. 3, pp. 195–208, 2008.
17. C. Bussler, "Analysis of the organization modeling capability of workflow-management-systems," in *PRIISM96 Conference Proceedings*, pp. 438–455, 1996.
18. O. M. G. (OMG), "Business process model and notation (bpmn) version 2.0," tech. rep., jan 2011.
19. W. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*, vol. 2. 2011.
20. 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.