Actionable Conformance Checking: From Intuitions to Code

Josep Carmona[1], Matthias Weidlich[2], and Boudewijn van Dongen[3]

[1] Universitat Politècnica de Cataluna, Spain,
jcarmona@cs.upc.edu,
[2] Humboldt University of Berlin, Germany
matthias.weidlich@hu-berling.de
[3] Eindhoven University of Technology, The Netherlands
B.F.v.Dongen@tue.nl

Abstract. Conformance checking is receiving increasing attention in the last years. This is due to several reasons, that can be summarized into two: the explosion of digital information that talks about processes, and the need to use this data in order to monitor and improve processes in organizations. Naturally, conformance checking addresses this by providing techniques capable of relating modeled and recorded process information. This paper overviews in a very accessible way the main techniques and feedback of the conformance checking field. Moreover, in order to make it actionable, code snippets are provided so that an organization can start a conformance checking project on its own data.

Key words: Conformance Checking, Process Mining, Business Process Management, BPMN, Petri nets, event logs, alignments

1 Introduction

Nowadays organizations are facing a digital transformation, that primarily requires active use of the tons of data available as a result of their operation. As processes are the main focus for the management of an organization, exposing processes to the data available helps to assess the alignment between observed and modeled behavior. When modeled and observed behavior are aligned, then one can be sure that the reality and the models describing it agree. In contrast, an organization may need to react in case of finding deviations between observed and modeled behavior. Conformance checking techniques [1] tackle this fundamental problem: to analytically asses the adequacy of a process model in representing the traces in an event log, extracting the deviations in case they exist. Due to the potential existence of regulations, guidelines, frauds and errors, conformance checking is becoming an essential element for an organization to prove the adherence to a desired behavior.

Conformance checking is a crucial dimension in process mining [2]: by relating modelled and observed behavior, process models that have either been discovered or manually created, can be confronted with event data. On its core, conformance

Carmona, J.; Weidlich, M.; Dongen, B. Actionable conformance checking: from intuitions to code. A: European Big Data Management & Analytics Summer School. "Big Data Management and Analytics, 9th European Summer School, eBISS 2019: Berlin, Germany, June 30–July 5, 2019: revised selected papers". Berlín: Springer, 2019, p. 1-24. ISBN 978-3-030-61627-4. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-61627-4_1

checking relies on the fundamental problem of identifying, among the set of runs of a process model (which can be infinite), the run that mostly resembles an observed trace.

In general, conformance checking has been applied to very different domains, including healthcare, banking, finance, transportation, manufacturing among others. The reader can see detailed use cases of all these fields in the web of the *IEEE Task Force on Process Mining*: `https://www.tf-pm.org`.

In this paper we aim at providing a gentle introduction to the conformance checking field, by describing its main techniques. Furthermore, we show code snippets illustrating some of the conformance checking techniques presented in this paper. The code snippets provided in this paper and related data is available in `https://github.com/matthiasweidlich/conf_tutorial/`.

2 Related Work

The field of conformance checking is relatively new. The definition of the area and a proposal of initial algorithms was presented in the scope of Anne Rozinat's PhD thesis at the TU/e [3] and corresponding publications [4, 5, 6, 7]. Important notions arise from this work, like *fitness* or *appropriateness* between a process model and log. Also, important algorithms result from this work, including the techniques to evaluate fitness based on the replay of the traces and the missing/remaining/produced/consumed tokens. Also in the scope of the TU/e, the seminal work under the PhD thesis of Arya Adriansyah is crucial for formalizing the notion of *alignments* [8]. Several applications of alignments are explored in the related publications, like performance analysis [9, 10], high-level deviations [11], privacy analysis of user behaviour [12], and alignment-based precision metrics [13].

Another work that has been important for conformance checking is the log conformance analysis presented in the scope of Matthias Weidlich's PhD thesis [14]. The thesis introduces the concept of *behavioural profiles*, as a tailored abstraction for processes that allows comparing recorded and modelled behaviour.

3 Process Models and Event Logs

Process models and event logs represent different conceptualizations of processes. When describing a process, a process model provides an abstraction, capturing some of the process' activities by means of *tasks*. A specific instance of a process, i.e., a case, then corresponds to a sequence of task executions, denoted *run*. In contrast, event logs store the executions of a certain process in a organization. In the remainder of this section we informally introduce these two conceptualizations with the help of a real-life example.

A process model that describes how a loan application is handled is illustrated in Fig. 1. This model is captured in the Business Process Model and Notation (BPMN). In BPMN, tasks are represented by rectangles; instantaneous events

are visualised by circles (in Fig. 1 they start or end the process); and execution dependencies are modelled by control flow arcs and diamond-shaped nodes, called gateways. The semantics of such a gateway determines the exact behaviour of a process, e.g., whether incoming arcs are synchronised (AND-gateway with a 'plus' symbol) or not (XOR-gateway with a 'cross' symbol); or whether outgoing arcs are enabled concurrently (AND-gateway) or mutually exclusive to each other (XOR-gateway). A run of the model (a sequence from start to end that agrees with the aforementioned semantics) is $\langle As, Aa, Fa, Sso, Ro, Ao, Aaa, Af \rangle$.

According to this model, a submitted application is either accepted or rejected, based on the aforementioned rules to check plausibility of the applicant's data. An accepted application is finalised by a worker, in parallel with the offer process. For each application, an offer is selected and sent to the customer. The customer reviews the offer and sends it back. If the offer is accepted, the process continues with the approval of the application and the activation of the loan. If the customer declines the offer, the application is also declined and the process ends. However, the customer can also request a new offer, in which case the offer is cancelled and a new offer is sent to the customer.

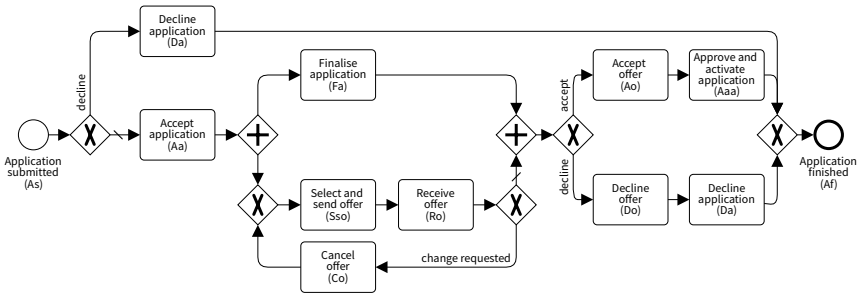

Fig. 1. Example process model of a loan application process in BPMN.

The recording of a single execution of an activity of a process in a information system is called an *event*. Typically, events are performed in a certain *context*, such as for example a specific loan application. This context is commonly given by the case as part of which an activity was executed. The notion of a case, therefore, binds together events, thereby allowing us to track the evolution of a case over time. The events related to a single case are called a *trace*.

The notion of a trace is fundamental for *event logs*. In essence, an event log is a collection of traces, each trace comprising events that can be sorted by their occurrence time. Consider for example our loan application process of Fig. 1. Tab. 1 shows an excerpt of such event log. The application with id $A5634$ is accepted by the system on January 1^{st} at 12:32 and the customer asks for a €2,000 loan. On January 3^{rd} the application is finalised and two days later, an offer is made to the customer for a €1,500 loan. The offer is received back on January 10^{th} and the customers did not sign it, nor did they indicate they want

Table 1. Example of a log of the loan application process.

Event	Application	Offer	Activity	Amount	Signed	Timestamp
...	...	...	...	...	...	...
e_{13}	$A5634$		Application submitted	€2,000		Jan 01, 12:31
e_{14}	$A5634$		Accept application	€2,000		Jan 01, 12:32
e_{15}	$A5635$		Application submitted	€5,000		Jan 02, 04:31
e_{16}	$A5635$		Accept application	€5,000		Jan 02, 04:32
e_{17}	$A5636$		Application submitted	€200		Jan 03, 06:59
e_{18}	$A5636$		Accept application	€200		Jan 03, 07:00
...	...	...	...	...	...	...
e_{22}	$A5634$		Finalise application			Jan 03, 09:00
e_{23}	$A5636$		Finalise application			Jan 03, 09:01
e_{24}	$A5635$		Decline application			Jan 03, 09:02
e_{25}	$A5635$		Decline application			Jan 03, 09:03
...	...	...	...	...	...	...
e_{30}	$A5636$	$O3521$	Select and send offer	€500		Jan 04, 16:32
...	...	...	...	...	...	...
e_{37}	$A5634$	$O3541$	Select and send offer	€1,500		Jan 05, 12:32
e_{38}	$A5636$	$O3521$	Receive offer		NO	Jan 05, 12:33
e_{38}	$A5636$	$O3521$	Cancel offer			Jan 05, 12:34
e_{39}	$A5636$	$O3542$	Select and send offer	€500		Jan 05, 13:29
e_{40}	$A5636$	$O3542$	Receive offer		YES	Jan 08, 08:33
e_{41}	$A5636$	$O3542$	Accept offer			Jan 08, 16:34
e_{42}	$A5634$	$O3541$	Receive offer		NO	Jan 10, 10:00
...	...	...	...	...	...	...
e_{54}	$A5634$	$O3541$	Decline offer			Jan 10, 10:04
...	...	...	...	...	...	...
e_{64}	$A5634$		Decline application			Jan 10, 10:05
e_{65}	$A5634$		Application finished			Jan 10, 10:06
e_{66}	$A5636$		Approve and activate application			Jan 10, 10:07
e_{67}	$A5636$		Application finished			Jan 10, 10:08
...	...	...	...	...	...	...

any changes. Therefore, a few minutes later, the offer is declined, which is also done for the application as a whole.

4 Conformance Checking

4.1 Quality Dimensions to Relate Process Models and Event Logs

By relating observed and modeled behavior, an organization can get insights on the execution of their processes with respect to the expectations as described in the models. If both process model M and event log L are considered as languages, their relation can be used to measure how good is a process model in describing the behavior recorded in an event log.

Hence, confronting M and L can help into understanding the complicate relation between modeled and recorded behavior. We now provide two visions of this relation, that represent two alternative perspectives: *fitness* and *precision*.

Fitness measures the ability of a model to explain the recorded execution of a process as recorded in an event log (see the example of Fig. 2 for an example of fitting behavior). It is the main measure to assess whether a model is well-suited to explain the recorded behaviour. To explain a certain trace, the process model is queried to assess its ability in replaying the trace, taking into account the control flow logic expressed in the model.

In general, fitness is the fraction of the behaviour of the log that is also allowed by the model. It can be expressed as follows.

$$fitness = \frac{|L \cap M|}{|L|} \tag{1}$$

Let us have a look at this fraction in more detail by examining the extreme cases. Fitness is 1, if the entire behaviour that we see in the log L is covered by the model M. Conversely, fitness is 0, if no behaviour in the log L is captured by the model M. In the Sect. 4.2 we will describe three different algorithms deriving artefacts that can be used to evaluate fitness.

We define a trace to be either *fitting* (it corresponds to a run of the model) or *non-fitting* (there is some deviation with respect to all runs of the model). For instance, the trace corresponding case $A5634$ in our running example is fitting, since there is a model run that perfectly reproduces this case, as shown in Fig. 2. In contrast, Fig. 3 shows the information for a trace that does not contain the event to signal that the application has been finalised (*Fa*).

Precision is the counterpart of fitness. It can be calculated by looking at the fraction of the model behaviour that is covered in the log.

$$precision = \frac{|L \cap M|}{|M|} \tag{2}$$

We see that precision shares the numerator in the fraction with fitness from (1). This implies that if we have a log and a model with no shared behaviour, fitness is zero, and by definition also precision is zero. However, the denominator is replaced with the amount of modelled behaviour.

In summary, for the two main metrics reported above, algorithms that can assess the relation between log and model need to be considered. In the next section, we describe the three main algorithmic perspectives to acccomplish this task.

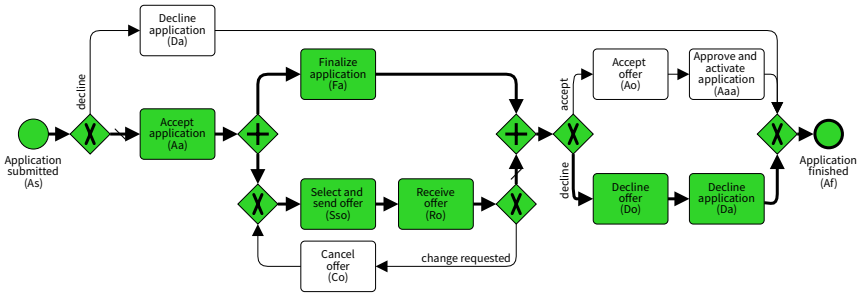

Fig. 2. Loan application process model with highlighted path corresponding to the fitting trace of case $A5634$ from the event log of Tab. 1.

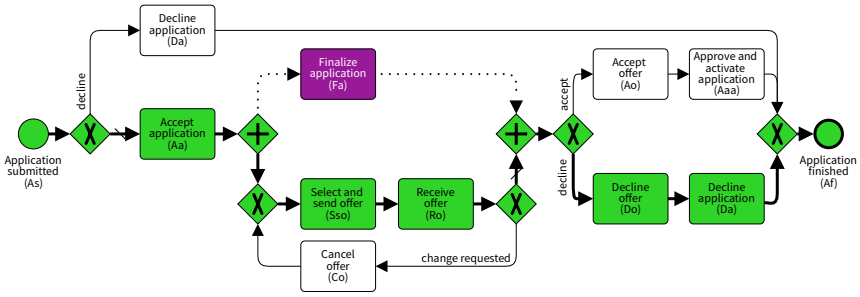

Fig. 3. Loan application process model with highlighted path corresponding to one trace, which does not include an event to signal that the application has been finalised (Fa). In magenta we show that the task (Fa) has not been observed but it is required to reach the final state of the process model.

4.2 Computing Conformance Checking Artefacts

The relation between a trace observed in the event log, and a process model, is described as a *conformance checking artefact*. In this section we will introduce three possible conformance checking artefacts, overviewed in Fig. 4. The reader is refered to [1] for a detailed explanation of the contents of this section.

Rule Checking The basic idea of rule-based conformance checking is to exploit rules that are satisfied by all the runs of a process model as the basis for analysis. Such rules define a set of constraints that are imposed by the process model. The verification of these constraints with respect to the traces of an event log, therefore, enables the identification of conformance issues.

Considering the running example of our loan application process as depicted in Fig. 1, rules derived from the process model include:

R1: An application can be accepted (Aa) at most once.
R2: An accepted application (Aa), that must have been submitted (As) earlier, and eventually an offer needs to be selected and sent (Sso) for it.

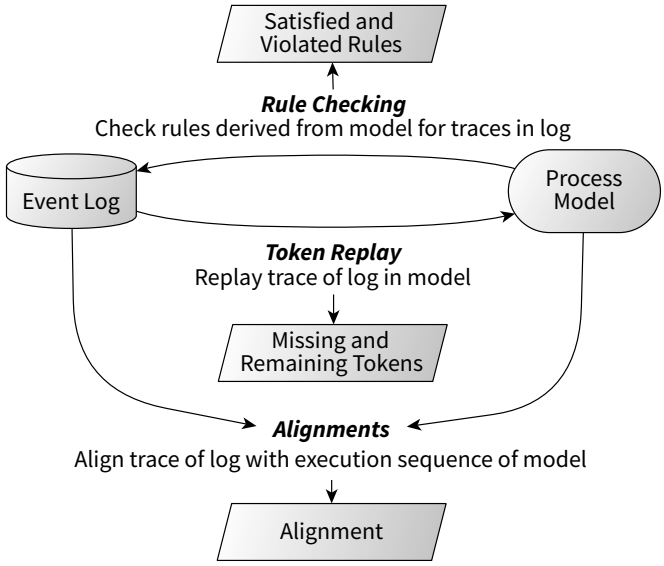

Fig. 4. General approaches to conformance checking and resulting conformance artefacts (from [1]).

R3: An application must never be finalised (Fa), if the respective offer has been declined (Do) already.
R4: An offer is either accepted (Ao) or declined (Do), but cannot be both accepted and declined.

A careful inspection of each one of the rules above would reveal that they are different in nature: rule R1 is an example of *cardinality rule*, which defines an upper and lower bound for the number of executions of an activity. Rule R2 contains a *precedence rule*, which establishes that the execution of a certain task is preceded by at least on execution of another task. Rule R3 establishes an *ordering rule*, whereas rule R4 represents an *exclusiveness rule*. Tables 2 and 3 show examples of cardinality and exclusiveness rules, respectively, for the running example and two log traces.

By assessing to what extent the traces of a log satisfy the rules derived from a process model, rule-based conformance checking focuses on the fitness dimension, i.e., the ability of the model to explain the recorded behaviour. Traces are fitting, if they satisfy the rules, or non-fitting if that is not the case. Let R_M be a predefined set of rules. Fitness can be defined according[1] to R_M:

$$\text{fitness}(L, M) = \frac{|\{r \in R_M \mid r \text{ is satisfied by all } t \in L\}|}{|R_M|} \tag{3}$$

[1] Notice that this makes fitness to depend on a particular set of rules, which is a limitation of the rule-based fitness checking.

Table 2. Precedence rules derived for the process model of the running example and their satisfaction (✓) and violation (✗) by the exemplary log trace $\langle As, Sso, Fa, Ro, Co, Ro, Aaa, Af \rangle$.

	As	*Da*	*Aa*	*Fa*	*Sso*	*Ro*	*Co*	*Ao*	*Aaa*	*Do*	*Af*
As											
Da	✓										
Aa	✓										
Fa	✓		✗								
Sso	✓		✗								
Ro	✓		✗		✓						
Co	✓		✗		✓	✓					
Ao	✓		✓	✓	✓	✓					
Aaa	✓		✗	✓	✓	✓		✗			
Do	✓		✓	✓	✓	✓					
Af	✓										

Table 3. Exclusiveness rules derived for the process model of the running example and their satisfaction (✓) and violation (✗) by the exemplary log trace $\langle As, Aa, Sso, Ro, Fa, Ao, Do, Da, Af \rangle$.

	As	*Da*	*Aa*	*Fa*	*Sso*	*Ro*	*Co*	*Ao*	*Aaa*	*Do*	*Af*
As	✓										
Da		✓						✗	✓		
Aa			✓								
Fa				✓							
Sso											
Ro											
Co											
Ao		✗						✓		✗	
Aaa		✓							✓	✓	
Do								✗	✓	✓	
Af											✓

As the reader may already have grasped, the dimension of precision is not targeted by rule-checking.

Token Replay Intuitively, this technique replays each trace of the event log in the process model by executing tasks according to the order of the respective events. By observing the states[2] of the process model during the replay, one can

[2] A state of a BPMN model is a distribution of tokens over the control flow arcs. A task is enabled in a state if its incoming control flow arc is assigned a token by the

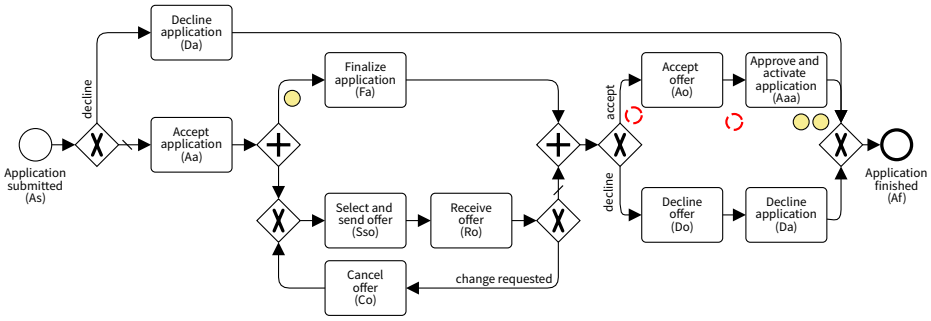

Fig. 5. State reached after replaying the full trace $\langle As, Aa, Sso, Ro, Ao, Aaa, Aaa\rangle$. One can see that there are three remaining tokens (denoted by yellow background), and two missing tokens (denoted by discontinous red lines).

determine whether, and to what extent, the trace indeed corresponds to a valid run of the process model.

In essence, token replay postulates that each trace in the event log corresponds to a valid execution sequence of the process model. This is verified by step-wise executing tasks of the process model, according to the order of the respective events in the trace. During this replay, we may observe two cases that hint at non-conformance (see Fig. 5):

(i) the execution of a task requires the consumption of a token on the incoming arc, but the arc is not assigned any token in the current state, i.e., a token is *missing* during replay;
(ii) the execution of a task produces a token at an outgoing arc, but this token is not consumed eventually, i.e., a token is *remaining* after replay.

By exploring whether the replay of a trace yields missing or remaining tokens, replay-based conformance checking mainly focuses on the fitness dimension. That is, the ability of the model to explain the recorded behaviour is the primary concern. Traces are fitting if their replay does not yield any missing or remaining tokens, and non-fitting otherwise:

$$\text{fitness}(L, M) = \frac{1}{2}\left(1 - \frac{\sum_{t\in L} missing(t, M)}{\sum_{t\in L} consumed(t, M)}\right) + \frac{1}{2}\left(1 - \frac{\sum_{t\in L} remaining(t, M)}{\sum_{t\in L} produced(t, M)}\right) \tag{4}$$

In contrast to rule checking, precision can be estimated using token replay [15], but unfortunately, the corresponding technique strongly relies on the assumption that traces are fitting; if they are not, then the estimation of precision through token replay can be significantly degraded [13].

respective distribution. If it executes, this token is *consumed*, i.e., no longer assigned to the arc. Moreover, a token is *produced* on the outgoing control flow arc of the task.

Alignments Alignments take a symmetric view on the relation between modelled and recorded behaviour. Specifically, they can be seen as an evolution of token replay. Instead of establishing a link between a trace and sequences of task executions in the model through replay, alignments directly connect a trace with a model run.

An alignment connects a trace of the event log with a run of the process model. It is represented by a two-row matrix, where the first row consists of activities as their execution is signalled by the events of the trace and a special symbol $\gg$ (jointly denoted by e_i below), and the second row consists of the activities that are captured by task executions of a run of the process model and a special symbol $\gg$ (jointly denoted by a_i):

log trace	e_1	e_2	...	e_n
model run	a_1	a_2	...	a_m

Each column in this matrix, a pair (e_i, a_i), is a *move* of the alignment, meaning that an alignment can also be understood as a sequence of moves. There are different types of such moves, each encoding a different situation that can be encountered when comparing modelled and recorded behaviour. We consider three types of moves:

- *Synchronous move*: A step in which the event of the trace and the task in the run correspond to each other. Synchronous moves denote the expected situation that the recorded events in the trace are in line with the tasks of a run of the process model. In the above model, a synchronous move means that it holds $e_i = a_i$ and $e_i \neq \gg$ (and thus $a_i \neq \gg$).
- *Model move*: When a task should have been executed according to the model, but there is no related event in the trace, we refer to this situation as a model move. As such, the move represents a deviation between the trace and the run of the process model in the sense that the execution of an activity has been skipped. In the above model, a model move is denoted by a pair (e_i, a_i) with $e_i = \gg$ and $a_i \neq \gg$.
- *Log move*: When an event in the trace indicates that an activity has been executed, even though it should not have been executed according to the model, the alignment contains a log move . Being the counterpart of a model move, a log move also represents a deviation in the sense of a superfluous execution of an activity. A log move is denoted by a pair (e_i, a_i) with $e_i \neq \gg$ and $a_i = \gg$.

Alignments are constructed only from these three types of moves (see an in-depth explanation on this in [1]).

For instace, let us use the running example (see Fig. 1) and the trace $\langle As, Aa, Sso, Ro, Ao, Aaa, Aaa \rangle$. A possible alignment with this trace is:

log trace	As	Aa	Sso	Ro	$\gg$	Ao	Aaa	Aaa	$\gg$
model run	As	Aa	Sso	Ro	Fa	Ao	Aaa	$\gg$	Af

This alignment comprises six synchronous moves, one log move, $(Aaa, \gg)$, and two model moves, $(\gg, Fa)$ and $(\gg, Af)$. The log move $(Aaa, \gg)$ indicates that the application had been approved and activated, even though this was not expected in the current state of processing (as this had just been done). The model move $(\gg, Fa)$ is the situation of the process model requiring that the application be finalised, which has not been done according to the trace. Furthermore, one can easily extract the original trace by projecting away the special symbol for skipping from the top row. Applying the projection to the bottom row yields the run of the model ($\langle As, Aa, Sso, Ro, Fa, Ao, Aaa, Af \rangle$).

In general, *optimal alignments*, i.e., alignments with minimal number of move or log moves, are preferred. The alignment shown above is optimal since there is no other alignment with least number of deviations. Computing (optimal) alignments is a hot research topic, which has been addressed in many papers in the last years [8, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]. In this paper, however, we will refrain from describing the state-of-the-art methods for alignment computation, and refer the interested reader to the aforementioned papers, or to [1].

Remarkably, alignments provide a simple means to quantify fitness. Again, this may be done based on the level of an individual trace or the event log as a whole. However, the aggregated cost of log moves and model moves may be a misleading measure, though, as it is not normalised. A common approach, therefore, is to normalise this cost by dividing it by the worst-case cost of a aligning the trace with the given model. Under a uniform assignment of costs to log and model moves, such a worst-case cost originates from an alignment in which each event of the trace T_i relates to a log move, whereas all task executions of a run σ of the model relate to a model move and σ is as short as possible. Since the cost induced by the model moves of a model run depends on its length, the shortest possible model run leading from the initial state to a final state in the model is considered for this purpose.

Realising the above idea, we obtain two ratios that denote the relative share of non-fitness in the alignments of a trace or an event log, respectively. Let M be a model and L an event log. Then, we denote by $cost(t, M)$ the cost of an optimal alignment of a trace $t \in L$ with respect to the model. Furthermore, let $cost(t, \langle\rangle)$ and $cost(\langle\rangle, x)$ be the costs of aligning a trace t with an empty model run, or some run $x \in M$ of the model with an empty trace, respectively. Then, fitness based on alignments is quantified for a trace or an event log:

$$\text{fitness}(L, M) = 1 - \left(\frac{\sum_{t \in L} cost(t, M)}{\sum_{t \in L} (cost(t, \langle\rangle)) + |L| \times \min_{x \in M} cost(\langle\rangle, x)} \right) \tag{5}$$

A simple precision metric based on alignments is grounded in the general idea of *escaping edges* [15]. To give the intuition, we assume that (i) the event log fits the process model; and (ii) that the process model is deterministic. The former means that we simply exclude non-fitting traces, for which the optimal alignment contains log moves or model moves, from the assessment of the precision of the

model. The latter refers to a process model not being able to reach a state, in which two tasks that capture the same activity of the process are enabled. The model of our running example (see Fig. 1) is deterministic.

For the activity of each event of a trace of the event log, we can determine a state of the process model right before the respective task would be executed. Under the above assumptions, this state is uniquely characterised. What is relevant when assessing precision, is the number of tasks enabled in this state of the process model. Let M be a process model and L an event log, with $t \in L$ as a trace and, overloading notation, $e \in t$ as one of the events of the trace. Then, by $\mathit{enabled}_M(e)$, we denote the number of tasks and, due to determinism of the process model also the number of activities that can be executed in the state right before executing the task corresponding to e.

Similarly, we consider all traces of the log that also contain events related to the activity of event e, say a, and have the same prefix, i.e., events that indicate that the same sequence of activities has been executed before an event signalling the execution of activity a. Then, we determine the number of activities for which events signal the execution directly after this prefix, i.e., the set of activities that have been executed in the same context as the activity a as indicated by event e. Let this number of activities be denoted by $\mathit{enabled}_L(e)$, which, under the above assumptions, is necessarily less than or equal to $\mathit{enabled}_M(e)$. Then, the ratio of both numbers captures the amount of 'escaping edges' that represent modelled behaviour that has not been recorded. As such, precision of log L and M is quantified as follows:

$$\text{precision}(L, M) = \frac{\sum_{t \in L, e \in t} \mathit{enabled}_L(e)}{\sum_{t \in L, e \in t} \mathit{enabled}_M(e)} \tag{6}$$

5 Code Snippets for Conformance Checking

In the previous section an informal introduction to conformance checking has been provided. Concepts like event log, process model, deviation, rule checking, token replay, alignment, fitness and precision should now be familiar to the reader. They are meant to define the complicate relation between modeled and recorded behavior. In this section we take the reader to practice, by introducing simple and intuitive Python code to make most of the aforementioned concepts actionable. Hopefully, the contents of this section can contribute to unleash the application of conformance checking.

5.1 Event Log Exploration

We start by providing examples on how to read an event log, and for extracting different types of information from it. The code provided is a subset of the one available in the repository used for this paper, where several other analyses can be found. The following code reads a log in XES format, the standard format for event logs approved by the IEEE [28].

```python
import xml.etree.ElementTree as et

def load_xes(file):
    log = []

    tree = et.parse(file)
    data = tree.getroot()

    # find all traces
    traces = data.findall('{http://www.xes-standard.org/}trace')

    for t in traces:
        trace_id = None

        # get trace id
        for a in t.findall('{http://www.xes-standard.org/}string'):
            if a.attrib['key'] == 'concept:name':
                trace_id = a.attrib['value']

        events = []
        for event in t.iter('{http://www.xes-standard.org/}event'):

            e = {'name': None, 'timestamp': None, 'resource': None,
            ↪  'transition': None}

            for a in event:
                e[a.attrib['key'].split(':')[1]] = a.attrib['value']

            events.append(e)

        # add trace to log
        log.append({'trace_id': trace_id, 'events': events})

    return log
```

Fig. 6. Code for reading an event log.

Once a log is read, one can extract valuable information from traversing it. For instance, the following code shows the length of the shortest and the longest trace in the log.

```python
log_file = 'conf_tutorial/financial_log.xes'
log = load_xes(log_file)
max_length = 0
min_length = 1000

for trace in log:
```

```
    if len(trace['events']) > max_length:
        max_length = len(trace['events'])

    if len(trace['events']) < min_length:
        min_length = len(trace['events'])

print('The longest trace contains %s events. The shortest trace: %s
↪  events.' %(max_length, min_length))
```

Also, the number of *trace variants*, i.e., number of different traces, of the log can be determined:

```
trace_list = []

for trace in log:
    events = []
    for event in trace['events']:
        events.append(event['name'])

    trace_list.append(tuple(events))

trace_variants = set(trace_list)

print('The log contains %s trace variants.' %len(trace_variants))
```

Events in the event log may have several attributes, like a timestamp or a resource. We can use these timestamps to compute the duration of a single trace. The following code returns the shortest and longest duration of all traces in the event log.

```
def get_timestamp(input_str: str):
    """
    Method to convert a string into a timestamp.

    :param input_str: timestamp as string
    """
    timestamp_format = '%Y-%m-%dT%H:%M:%S.%f%z'

    return datetime.strptime(''.join(input_str.rsplit(':', 1)),
    ↪  timestamp_format)

from datetime import datetime, timedelta


max_duration = timedelta(microseconds=1)
min_duration = timedelta(days=10000)

for trace in log:

```

```
    # we only need to consider the first and last event in the
    ↪ trace
    first_e = trace['events'][0]
    last_e = trace['events'][-1]

    t0 = get_timestamp(first_e['timestamp'])
    t1 = get_timestamp(last_e['timestamp'])
    duration = t1 - t0

    if duration > max_duration:
        max_duration = duration
    elif duration < min_duration:
        min_duration = duration

print('The shortest process instance took %s; the longest %s'
↪ %(min_duration, max_duration))
```

As a final illustration of event log exploration, we focus on another event attribute. In the following code, we output how many different resources are used across the process instances, and the ratio of events that are processed by a resource.

```
resources = []

for trace in log:
    for event in trace['events']:
        resources.append(event['resource'])

print('All process instances use %s different resource in total' %
↪ len(set(resources)))
no_res = resources.count(None)
print('%.2f%% of all events are processed by a resource.'
↪ %(((len(resources)-no_res)/len(resources))*100 ))
```

5.2 The Computation of Conformance Checking Artefacts

We now consider how conformance checking artefacts can be computed so that deviations between modeled and recorded behavior can be obtained.

Process models will be assumed to be defined as Petri nets. In the repository provided with this paper, a Petri net Python class (denoted `PetriNet` in the code) will be used, which contains the standard helper functions to manage it. We assume the reader to be familiar with Petri nets in this paper (if not, a nice tutorial can be found in [29]). The following code reads a process model for the running example, sets the initial state, and finally draws it.

```
%run ./conf_tutorial/pn.py

net = PetriNet()
```

```
load(net, "./conf_tutorial/financial_log_80_noise.pnml")

# mark the initial place
net.add_marking(1,1)
# visualise it
draw_petri_net(net)
```

Importantly, mapping events in the event log and tasks in the process model is an important step so that the conformance checking artefacts can be computed. The following code sets up some helper dictionaries to relate Petri net transition IDs and activity labels in the event log to each other. Observe that for the sake of simplicity, an activity label is only assigned to a single transition. However, multiple transitions may carry a τ label, representing a silent transition (a transition that does not correspond to any event in the log).

```
# helper mappings between ids and labels
mapping = net.get_mapping()
rev_mapping = {}
for k, v in net.get_mapping().items():
    for k2 in v:
        rev_mapping[k2] = k

from pprint import pprint
# mapping from labels to LISTS of transitions ids
pprint(mapping)

# mapping from transitions id to label
pprint(rev_mapping)
```

The next code illustrates how, given an initial marking, the currently enabled transitions may be identified, how the marking is changed by firing a transition, and how the marking may be adapted to enable a transition.

```
print("Initial marking: ", net.get_marking())

enabled = net.all_enabled_transitions()
print("Enabled transitions in initial marking: ",
      list(map((lambda k: rev_mapping[k]), enabled)))

# Fire enabled transition (take the first, but there is only one)
net.fire_transition(enabled[0])
enabled = net.all_enabled_transitions()
print("Enabled transitions after firing first transition: ",
      list(map((lambda k: rev_mapping[k]), enabled)))

# Check whether the transition with label 'O_CREATED' is enabled
# (there is only one transition carrying this label)
print("Is transition 'O_CREATED' enabled?",
```

```
      net.is_enabled(net.get_mapping()['O_CREATED'][0]))

# Enable the transition by changing the marking and adding tokens to the input
# places of the transition with label 'O_CREATED'
input_places = net.get_input_places(net.get_mapping()['O_CREATED'][0])

for p in input_places:
    net.add_marking(p,1)

# Again, check whether the transition with label 'O_CREATED' is enabled
print("Is transition 'O_CREATED' enabled after tokens have been added to the places in its preset?",
      net.is_enabled(net.get_mapping()['O_CREATED'][0]))

# Check whether further transitions have been enabled by adding the token to
# the places in the preset of the transition with label 'O_CREATED'
enabled = net.all_enabled_transitions()
print("Enabled transitions after adapting the marking: ",
      list(map((lambda k: rev_mapping[k]), enabled)))

print("Current marking: ", net.get_marking())
```

We are now ready to define and use conformance checking artefacts. We will start with rule checking. Specifically, we consider a cardinality rule that checks a lower and an upper bound for the number of executions of an activity for a particular trace, as well as an ordering rule that checks whether executions of one activity happen only after executions of another activity.

More concretely, we check whether the five most frequent trace variants satisfy the following rules:

1. The application is completed at least once (activity "W_Completeren aanvraag").
2. The application is submitted at most once (activity "A_SUBMITTED").
3. The income lead ("W_Afhandelen leads") is fixed only after the preacceptance ("A_PREACCEPTED"), but never before.

```
def check_lower_bound(trace: [], act: str, bound: int) -> bool:
    count = trace.count(act)
    return count >= bound

def check_upper_bound(trace: [], act: str, bound: int) -> bool:
    count = trace.count(act)
    return count <= bound

```

```
def check_order_after(trace: [], act_1: str, act_2: str) -> bool:
    if act_1 not in trace or act_2 not in trace:
      return True
    idx_1 = [i for i, x in enumerate(trace) if x == act_1]
    idx_2 = [i for i, x in enumerate(trace) if x == act_2]
    return idx_1[0] >= idx_2[-1]

# compute the trace variants sorted by frequency
trace_variants = {}
for trace in log:
    events = []
    for event in trace['events']:
        events.append(event['name'])
    trace_variants[tuple(events)] =
    ↪  trace_variants.get(tuple(events), 0) + 1
trace_variants_sorted_by_freq = sorted(trace_variants.items(),
↪  key=lambda kv: kv[1], reverse=True)

for k in range(5):
    trace_k = list(trace_variants_sorted_by_freq[k][0])
    print("Checking trace: %s" % trace_k)
    print("Application completed at least once? ",
    ↪  check_lower_bound(trace_k, 'W_Completeren aanvraag', 1))
    print("Application submitted at most once? ",
    ↪  check_upper_bound(trace_k, 'A_SUBMITTED', 1))
    print("Fixing income lead only after preaceptance? ",
    ↪  check_order_after(trace_k, 'W_Afhandelen leads',
    ↪  'A_PREACCEPTED'))
```

We can also apply token replay on the running example. The following code illustrates how to do token replay for a trace, and how to evaluate fitness for the 30 most frequent variants of the event log.

```
def replay_trace(net: PetriNet, trace: []) -> (int, int, int, int):
    produced = 1
    consumed = 1
    missing = 0

    # replay trace, event by event
    for event in trace:
        # identify transition, assumption here is that there is only
        ↪  one transition for the label
        transition = net.get_mapping()[event][0]
        # check if the transition is enabled
        if not net.is_enabled(transition):
            # not enabled, so add a token to all input places that
            ↪  are not marked
            for p in net.get_input_places(transition):
                if net.marking[net.index_of_place(p)] == 0:
```

```
                    # record the token as missing
                    missing += 1
                    net.add_marking(p, 1)

        # record the numbers produced and consumed tokens when
        ↪  firing the transition
        produced += len(net.get_input_places(transition))
        consumed += len(net.get_output_places(transition))
        net.fire_transition(transition)

    # we expect one token left, everything else counts as remaining
    remaining = sum(net.get_marking()) - 1
    return produced, consumed, missing, remaining


def fitness(net: PetriNet, log_freq: dict) -> float:
    sum_prod = 0
    sum_cons = 0
    sum_miss = 0
    sum_rema = 0

    for trace_var, freq in log_freq.items():
        # keep copy of marking
        marking = list(net.get_marking())
        # replay trace
        replay_values = replay_trace(net, trace_var)
        sum_prod += log_freq[trace_var] * replay_values[0]
        sum_cons += log_freq[trace_var] * replay_values[1]
        sum_miss += log_freq[trace_var] * replay_values[2]
        sum_rema += log_freq[trace_var] * replay_values[3]
        # restore marking
        for k,v in net.places.items():
            net.add_marking(v, marking[k])

    return 0.5 * (1 - sum_miss / sum_cons) + 0.5 * (1 - sum_rema /
    ↪  sum_prod)

fitness_value = 0
for k in range(30):
    log_k = {t[0]:t[1] for t in
    ↪  trace_variants_sorted_by_freq[k:k+1]}
    log_x = {t[0]:t[1] for t in
    ↪  trace_variants_sorted_by_freq[0:k+1]}
    fitness_value_k = fitness(net, log_k)
    fitness_value = fitness(net, log_x)
    print("Fitness value of the single %s-most frequent trace
    ↪  variant: %f" % (k+1, fitness_value_k))
    print("Fitness value of %s-most frequent trace variants: %f" %
    ↪  (k+1, fitness_value))
```

Finally, we provide code to illustrate how alignments can be also used as conformance checking artefact. The following code illustrates how to use them to show deviations. We will be using the alignment functionality that is contained in the Python class `Astar`, provided also in the repository of this paper. In the following code snippets, we will use some of the objects computed before, like the Petri net, and the most frequent variants in the event log.

```
from pprint import pprint
%run ./conf_tutorial/alignment.py

# select some most frequent traces
traces = dict()
for k in range(10):
    traces[k] = list(trace_variants_sorted_by_freq[k][0])

# capture details on which places denote the start and the end of
↪  the process model
index_place_start = 0
index_place_end = 1

# run alignment construction
a = Astar()
alignments = a.Astar_Exe(net, traces, index_place_start,
↪  index_place_end, no_of_solutions=1)

# print the alignments
for k,t in traces.items():
  print('Trace in the log: ', t)
  print('Optimal alignment: ')
  pprint(alignments[k][0])
```

And now alignment-based fitness can be reported, as illustrated in the code below:

```
def fitness(net: PetriNet, alignments: list, log_freq: list) ->
↪  float:
    # the shortest model run in our example contains seven elements
    shortest_seq_in_net = 7

    async_moves = 0
    max_cost = 0

    for k in range(len(alignments)):
        async_moves += log_freq[k] * len([x for x in alignments[k]
        ↪  if (x[0] == '-' or x[1] == '-')])
        max_cost += log_freq[k] * (shortest_seq_in_net +
        ↪  len(alignments[k]))

    return round(float(async_moves) / float(max_cost), 3)
```

```
13
14 fitness_value = 0
15 for k in range(len(alignments)):
16     alignments_simple_k = [alignments[k][0]]
17     alignments_simple = [alignments[x][0] for x in range(k+1)]
18     log_freq_k = [trace_variants_sorted_by_freq[k][1]]
19     log_freq = [trace_variants_sorted_by_freq[x][1] for x in
       ↪  range(k+1)]
20     fitness_value_k = fitness(net, alignments_simple_k, log_freq_k)
21     fitness_value = fitness(net, alignments_simple, log_freq)
22     print("Fitness value of the single %s-most frequent trace
       ↪  variant: %f" % (k+1, fitness_value_k))
23     print("Fitness value of %s-most frequent trace variants: %f" %
       ↪  (k+1, fitness_value))
```

6 Concluding Remarks

In this paper we have introduced conformance checking and provided code snippets to make the discipline actionable in practice. The paper focuses in the definition and use of the main conformance checking artefacts, namely rule checking, token replay and alignments, so that a clear insight on the relation between modeled and observed behavior can be obtained from them.

To make it accessible, we have chosen to stay on simple, specially tailored, Python code that is sufficient for the main purpose of this paper. For the reader that became interested, we strongly advice to look for other open-source scripting libraries that can be also used to make conformance checking and process mining actionable: PMLAB [30], BupaR [31], pm4py [32] are some examples.

Acknowledgments. This work has been supported by MINECO and FEDER funds under grant TIN2017-86727-C2-1-R.

References

1. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking - Relating Processes and Models. Springer (2018)
2. van der Aalst, W.M.P.: Process Mining - Data Science in Action, Second Edition. Springer (2016)
3. Rozinat, A.: Process Mining Conformance and Extension. PhD thesis, Technische Universiteit Eindhoven (2010)
4. Rozinat, A., van der Aalst, W.M.P.: Conformance testing: Measuring the fit and appropriateness of event logs and process models. In: Business Process Management Workshops, BPM 2005 International Workshops, BPI, BPD, ENEI, BPRM, WSCOBPM, BPS, Nancy, France, September 5, 2005, Revised Selected Papers. (2005) 163–176

5. van der Aalst, W.M.P., Dumas, M., Ouyang, C., Rozinat, A., Verbeek, H.M.W.E.: Choreography conformance checking: An approach based on BPEL and Petri nets. In: The Role of Business Processes in Service Oriented Architectures, 16.07. - 21.07.2006. (2006)
6. van der Aalst, W.M.P., Dumas, M., Ouyang, C., Rozinat, A., Verbeek, H.M.W.E.: Conformance checking of service behavior. ACM Trans. Internet Techn. **8**(3) (2008) 13:1–13:30
7. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. **33**(1) (2008) 64–95
8. Adriansyah, A.: Aligning observed and modeled behavior. PhD thesis, Technische Universiteit Eindhoven (2014)
9. Adriansyah, A., Buijs, J.C.A.M.: Mining process performance from event logs. In: Business Process Management Workshops — BPM 2012 International Workshops, Tallinn, Estonia, September 3, 2012. Revised Papers. (2012) 217–218
10. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery **2**(2) (2012) 182–192
11. Adriansyah, A., van Dongen, B.F., Zannone, N.: Controlling break-the-glass through alignment. In: International Conference on Social Computing, SocialCom 2013, SocialCom/PASSAT/BigData/EconCom/BioMedCom 2013, Washington, DC, USA, 8–14 September, 2013. (2013) 606–611
12. Adriansyah, A., van Dongen, B.F., Zannone, N.: Privacy analysis of user behavior using alignments. Information Technology **55**(6) (2013) 255–260
13. Adriansyah, A., Munoz-Gama, J., Carmona, J., van Dongen, B.F., van der Aalst, W.M.P.: Measuring precision of modeled behavior. Inf. Syst. E-Business Management **13**(1) (2015) 37–67
14. Weidlich, M.: Behavioural Profiles : A Relational Approach to Behaviour Consistency. doctoral thesis, Universität Potsdam (2011)
15. Munoz-Gama, J., Carmona, J.: A fresh look at precision in process conformance. In: Business Process Management — 8th International Conference, BPM 2010, Hoboken, NJ, USA, September 13–16, 2010. Proceedings. (2010) 211–226
16. van Dongen, B.F.: Efficiently computing alignments - using the extended marking equation. In: Business Process Management - 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9-14, 2018, Proceedings. (2018) 197–214
17. Taymouri, F., Carmona, J.: Model and event log reductions to boost the computation of alignments. In: Proceedings of the 6th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2016), Graz, Austria, December 15-16, 2016. (2016) 50–62
18. de Leoni, M., Marrella, A.: Aligning real process executions and prescriptive process models through automated planning. Expert Syst. Appl. **82** (2017) 162–183
19. Reißner, D., Conforti, R., Dumas, M., Rosa, M.L., Armas-Cervantes, A.: Scalable conformance checking of business processes. Paper submitted to "International Conference on Business Process Management (BMP 2017)" in Barcelona, Spain. (March 2017)
20. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Scalable process discovery and conformance checking. Software and System Modeling **17**(2) (2018) 599–631
21. García-Bañuelos, L., van Beest, N.R., Dumas, M., Rosa, M.L., Mertens, W.: Complete and interpretable conformance checking of business processes. IEEE Transactions on Software Engineering **44**(3) (March 2018) 262–290

22. Taymouri, F., Carmona, J.: A recursive paradigm for aligning observed behavior of large structured process models. In: 14th International Conference of Business Process Management (BPM), Rio de Janeiro, Brazil, September 18 - 22. (2016)
23. Taymouri, F., Carmona, J.: An evolutionary technique to approximate multiple optimal alignments. In: Business Process Management - 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9-14, 2018, Proceedings. (2018) 215–232
24. van Dongen, B., Carmona, J., Chatain, Th., Taymouri, F.: Aligning modeled and observed behavior: A compromise between complexity and quality. In Dubois, E., Pohl, K., eds.: Proceedings of the 29th International Conference on Advanced Information Systems Engineering (CAiSE'17). Volume 10253 of Lecture Notes in Computer Science., Essen, Germany, Springer (June 2017) To appear.
25. Bloemen, V., van de Pol, J., van der Aalst, W.M.P.: Symbolically aligning observed and modelled behaviour. In: 18th International Conference on Application of Concurrency to System Design, ACSD 2018, Bratislava, Slovakia, June 25-29, 2018. (2018) 50–59
26. Taymouri, F., Carmona, J.: Structural computation of alignments of business processes over partial orders. In: 19th International Conference on Application of Concurrency to System Design, ACSD 2019, Aachen, Germany, June 23-28, 2019. (2019) 73–81
27. Padró, L., Carmona, J.: Approximate computation of alignments of business processes through relaxation labelling. In: Business Process Management - 17th International Conference, BPM 2019, Vienna, Austria, September 1-6, 2019, Proceedings. (2019) 250–267
28. Acampora, G., Vitiello, A., Stefano, B.N.D., van der Aalst, W.M.P., Günther, C.W., Verbeek, E.: IEEE 1849: The XES standard: The second IEEE standard sponsored by IEEE computational intelligence society [society briefs]. IEEE Comp. Int. Mag. **12**(2) (2017) 4–8
29. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4) (April 1989) 541–574
30. Carmona, J., Solé, M.: PMLAB: an scripting environment for process mining. In: Proceedings of the BPM Demo Sessions 2014 Co-located with the 12th International Conference on Business Process Management (BPM 2014), Eindhoven, The Netherlands, September 10, 2014. (2014) 16
31. Janssenswillen, G., Depaire, B., Swennen, M., Jans, M., Vanhoof, K.: bupar: Enabling reproducible business process analysis. Knowl.-Based Syst. **163** (2019) 927–930
32. Berti, A., van Zelst, S.J., van der Aalst, W.M.P.: Process mining for python (pm4py): Bridging the gap between process- and data science. CoRR **abs/1905.06169** (2019)