UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Karl-Kristjan Kokk

Compliance Monitoring of Data-Aware Declarative Process Models

Master's Thesis (30 ECTS)

Supervisor: Fabrizio Maria Maggi, PhD

Tartu 2020

Compliance Monitoring of Data-Aware Declarative Process Models

Abstract:
Runtime Compliance Monitoring in Business Process Management is an area that is dedicated to the early detection of non-conformance between a set of business constraints and the events that are recorded by an Information System supporting the execution of a business process. Companies need to make sure that their business practices conform to predefined requirements so that, in case of non-conformance, it is possible to take appropriate action to mitigate any further damage. Runtime monitoring makes it possible to continuously check the state of a process execution w.r.t to a set of business constraints.

This thesis focuses on a Runtime Compliance Monitoring technique that is able to check along with constraints over the control flow of a process also constraints over the data perspective of a business process. For modeling business constraints Multi-Perspective Declare (MP-Declare) is used. Alloy model checker is used for checking individual constraints at runtime as well as possible conflicts between two or more business constraints. The approach developed in this thesis was evaluated with synthetic and real life event logs and it was also compared against another existing approach. This implementation is also integrated as part of a process mining application.

Keywords: Process Mining, Alloy, Business Process, MP-Declare, Declarative Model, Data-aware, Runtime Compliance Monitoring

CERCS: P170 Computer Science, Numerical Analysis, Systems, Control

Andmetoega deklaratiivsete protsesside mudelite käitusaegne seire

Lühikokkuvõte:
Käitusaegne seire äri protsesside juhtimises on valdkond, mis on pühendunud selle, et varakult tabada mittevastavused ärimudeli ja toimunud sündmuste vahel, mis on salvestatud ettevõtte äriprotsesse toetava infosüsteemi poolt. Ettevõtetel on vaja kindel olla, et nende äripraktikad on vastavuses eelnevalt defineeritud nõuetele ning juhul kui on mitte vastavus, siis on võimalik võtta vastu otsuseid, mis leevendavad tulevast kahju. Käitusaegne seire võimaldab pidevalt jälgida, ärinõuete seisu.

Käesolev lõputöö keskendub käitusaegse seire tehnikale, mis võimaldab lisaks äriprotsessi kontrollvoo jälgimisele ka jälgida andmete nõudeid äriprotsessile. Ärinõuete mudeldamise jaoks on kasutatud andmetoega Declare keelt. Mudeli õiguse kontrollimiseks kasutatakse Alloy'd, millega on võimalik jälgida üksikuid ärireegleid ja ka võimalikke konflikte kahe või rohkema ärireegli vahel. Arendatud seireviis on testitud sünteetiliste, päris elu logide peal ning võrreldud ka teise olemasoleva seireviisiga. Käsitletud tehnika on integreeritud protsessikaeve rakendusteeki.

Võtmesõnad: Protsessikaeve, Alloy, Äriprotsess, MP-Declare, Deklaratiivne mudel, Andmetugi, Käitusaegne seire

CERCS: P170 Arvutiteadus, Arvutusmeetodid, Süsteemid, Juhtimine

Contents

1 Introduction

Companies, organizations and institutions need to make sure that their processes comply with given regulations. Processes that violate those regulations may cause reputational and financial damage to the company or, in the worst case scenario, the violations lead to severe penalties and even lawsuits against the company [1].

Compliance monitoring is a functionality that every process-aware information system should have and it is the ability to verify at runtime whether the monitored progress of process instances is compliant with a given process model [2]. Process models are either imperative or declarative. In imperative models such as BPMN and Petri nets all possible execution paths are modeled. These modeling languages are good for describing processes with low variability. Declarative models such as Declare are more suitable for complex modeling because the modeler can specify the execution paths as a set of constraints where any behavior is allowed if it does not conflict with any constraints.

Most of the compliance monitoring techniques use as a reference model a set of business constraints and monitor only the control flow of the process which, i.e., check if activities are executed in the correct order. One of the drawbacks of a control flow based technique is that it does not take into consideration the data perspective of a process execution, which may contain information such as what kind of resource was used, what kind of data was manipulated and who was executing the activity. Data-aware or multi-perspective compliance monitoring approaches use data attributes along with the control-flow to specify the reference set of business constraints. Deviations where the ordering of the activities is correct but the data values associated with activities are wrong will not be revealed by control-flow based compliance monitoring. However, with a data-aware approach it is possible to discover these non-conformities.

In this thesis, we propose an approach for compliance monitoring based on Multi-Perspective Declare models using the Alloy model checker. Multi-Perspective Declare (MP-Declare) is a well-known data-aware declarative process modeling language [3]. Alloy can produce a process execution that complies with certain specifications. In our case, a given partial trace is compliant with a set of MP-Declare constraints if Alloy is able to find a completion of the partial trace compliant with the set of constraints. This approach is able to identify whether a single constraint is violated and it is also able to identify if there is a conflict between two or more constraints. This application requires two inputs. The first one is an MP-Declare model and the other one is a real-time stream of process executions from an Information System supporting the execution of a business process which in this thesis will be simulated by a log streamer. The incoming data is processed by the core module of our application and the results are shown instantaneously on a user interface. This technique has been tested using synthetic and real life event logs and has been compared to another state-of-the art compliance monitoring technique. This solution is implemented as part of RuM [4], a novel application for rule mining.

The structure of this thesis is the following: Section 2 gives an overview of the existing literature about this subject. Section 3 gives an overview of the main concepts that are used in this thesis. In Section 4, the problem is explained in detail, the research questions are formulated and it is also specified how they are going to be answered. In Section 5, the approach that was developed for this thesis is discussed. It includes how the model and partial traces are encoded in Alloy. In Section 6, the implementation of the proposed approach is presented. In Section 7, an evaluation of the proposed technique and a comparison with another technique is reported. Section 8 concludes the thesis and provides suggestions for future work.

2 Background

In this section, we explain some of the concepts used in this thesis.

2.1 Process Mining

Process mining is a research field that deals with process discovery, process monitoring and process enhancement by leveraging data from event logs. The event logs are usually created by Process-Aware information systems (PAIS) supporting the execution of business processes. Process discovery takes an event log as an input and generates a model without any a-priori knowledge. Conformance checking takes a process model and an event log as inputs and checks whether there are discrepancies between the log and the model. The latter will be discussed further in this thesis. In process enhancement, an existing process model is improved by using information recorded in the event log [5].

2.2 Event log

An event log represents the execution history of a business process. It is made up of traces. Each trace represents an execution of a process. A trace consists of events which are ordered sequentially. To store data which accompanies traces or events there are attributes in key-value pairs. Each event commonly has an activity name and a timestamp and traces usually have an ID attribute attached to it. In addition to the trace and event attributes mentioned before traces and events can also have other numerical or string attributes.

Event logs can be stored as eXtensible Event Stream files (XES) [6]. In Listing 1 an extract of an XES event log which was created using a synthetic log generator [7] is given. The example shows a full representation of a trace where 3 events are stored. The event log data starts with a <log> tag. The next level is a <trace> tag which marks the start of the trace. Under the "concept:name" trace attribute the trace ID is stored. Under the <event> tag there is also a "concept:name" field which refers to the name of the activity and the "time:timestamp" field which refers to the time when the event took place. There are also additional string and numerical event attributes provided such as "TransportType" which has value "Car" and "Price" which has value "47.15" under the "BookTransport" event.

A prefix of a trace is the sub-trace where all the events up to a specified index are included. For example in Listing 1 the prefix with index 1 would be the sub-trace composed of one event "ApplyForTrip" and prefix with index 2 would be the sub-trace containing events "ApplyForTrip" and "BookTransport". The suffix of a trace are all the events that would not be included in the trace prefix.

```
<trace>
  <string key="concept:name" value="Case No. 1"/>
  <event>
    <string key="concept:name" value="ApplyForTrip"/>
    <string key="lifecycle:transition" value="complete"/>
    <date key="time:timestamp" value="2019-03-27T19:20:09+02:00"/>
  </event>
  <event>
    <string key="TransportType" value="Car"/>
    <string key="concept:name" value="BookTransport"/>
    <string key="lifecycle:transition" value="complete"/>
    <float key="Price" value="47.15"/>
    <date key="time:timestamp" value="2019-03-27T20:10:56+02:00"/>
  </event>
  <event>
    <string key="concept:name" value="BookAccomodation"/>
    <string key="lifecycle:transition" value="complete"/>
    <float key="Price" value="13.53"/>
    <date key="time:timestamp" value="2019-03-27T20:21:56+02:00"/>
  </event>
</trace>
```

Listing 1. XES Example

2.3 Linear Temporal Logic

Linear Temporal Logic or LTL for short [8] is a modal logic where it is possible to encode a formula about the structure of a sequence. In order to understand Table 3 and the operators used under the column "LTL semantics" the explanations and representation about future and past LTL operators are given in Table 1 and 2 respectively.

Operator	**Semantics**
$F\psi_1$	ψ_1 holds true now or somewhere in the future
$X\psi_1$	ψ_1 holds true in the next position
$G\psi_1$	ψ_1 holds true now and in every position in the future
$\psi_1 U \psi_2$	ψ_2 will hold true but until that moment ψ_1 will hold true

Table 1. Future LTL operators

2.4 Declarative process models

There are two types of process models which are imperative and declarative. In imperative process modeling, it is required to model all the possible execution paths. If there is

Operator	Semantics
$O\psi_1$	ψ_1 held true sometime in the past
$Y\psi_1$	ψ_1 held true in the previous position
$\psi_1 S \psi_2$	sometime in the past ψ_2 held true and after that until now ψ_1 held true

Table 2. Past LTL operators

a path that is not specified, then that path is not allowed. Petri nets [9] and Business Process Modeling Notation (BPMN) [10] are two main examples of imperative process modeling languages.

In declarative process modeling, constraints are specified and if an execution path does not violate the specified constraints then the execution path is allowed. An example of declarative process modeling language which is also used in this thesis is Declare [11]. In Declare, constraints are instances of predefined templates. Templates, in turn, are abstract entities that define parameterized classes of properties. Declare constraints can be specified through LTL over finite traces. Templates with their description and LTL-semantics are summarized in Table 3.

There are four categories of Declare templates: existence, relation, negative relation and choice. Existence templates are init, absence, existence and exactly. These are all unary templates which mean that they only have one activity as a parameter. In the relation group there are responded existence, response, precedence, alternate response and alternate precedence, chain response and chain precedence. These templates show a dependency between two activities. The third group is negative relation. These templates disallow the execution of certain activities. These constraints are all the constraints from Table 3 with a word "not" in it. The last group is choice where one of the two activities has to occur. Choice and exclusive choice constraints belong to the latter constraint category.

The response constraint with the LTL semantics G(A $\Rightarrow$ FB) is satisfied in trace $t_1 = \langle A, A, B, C \rangle$ and vacuously (or trivially) satisfied [12] in $t_2 = \langle B, B, C, D \rangle$ because the activation activity A never occurs. This constraint is however violated in trace $t_3 = \langle A, B, A, C \rangle$ because after the second activity A the activity B never occurs.

Single perspective conformance checking means that only the control-flow is being looked at. A constraint is activated when an activation activity occurs. When a constraint is activated the occurrence of an activation imposes an obligation on the execution of a second activity (called target). Activity A is an activation for the constraint Response(A, B) and activity B is the target. When activity A is executed this means that in order for the constraint to be satisfied the target activity must occur eventually after A. Otherwise the constraint will be violated. In trace t_1 the activation occurs and is satisfied however in t_3 the second activation activity leads to a violation because the second target activity

does not occur.

There can be cases where constraints are in conflict and cases where it is impossible to have any traces that would conform to the set of constraints. When we have constraint Response(A, B) and Not response(A,B) then these constraints can be in conflict with each other if A occurs but compliant traces can still be found like t_2 for example because here the constraints are never activated. But if a constraint Init(A) is added to the model then no compliant traces can be found because in this case activity A is forced to occur.

Multi-perspective conformance checking means that along with control-flow the data flow will also be looked at. While Declare only allows to describe the control-flow of a process, its extension MP-Declare allows to specify the process behavior from the perspective of data. Here, constraints are specified over traces but the constraints can have requirements on the payload of an activity, i.e., on the set of pairs attribute-value associated to the activity when it occurs. MP-Declare specifies 3 possible types of conditions on the payload of an activity.

The first type of condition is activation condition which is checked when an activation occurs and only if the activation condition holds then the constraint becomes active. For example consider the trace from Listing 1 and a constraint Absence(BookTransport). With the given trace this condition is violated, but if this constraint would now also have an activation condition like Absence(BookTransport)[price > 50] which would mean that Absence(BookTransport) constraint is activated when the price is over 50, since the price is 47,15, then this constraint will not be activated and therefore will be (vacuously) satisfied.

The second type of condition is correlation condition. This condition must be satisfied when the target occurs and the constraint is activated. This condition specifies the payload relationship between activation and target. Note that this condition is not applicable to existence template category since they have an activation activity only. Here we can have for example ChainResponse(BookTransport, BookAccomodation)[A.TransportType == "Car"] | [T.price < 15]. This means that when BookTransport activity occurs with TransportType equal to "Car" (A means Activation) this must be immediately followed by a BookAccomodation activity where the price is lower than 15 (T means Target). In the case of the given trace example from Listing 1 the activation condition activates the constraint and the correlation condition holds.

The third type of condition is time condition. This is used to specify a time distance between activation and target. A time condition has the specific format which is start, end, unit. Unit is used to specify whether the time distance is expressed in seconds (s), minutes (m), hours (h) or days (d). An example can be Response(BookTransport, BookAccomodation)[A.TransportType == "Car"] | [B.price < 15] | [1,3,d]. This can be interpreted as when a BookTransport occurs with TransportType "Car" then after more than 1 day and less than 3 days BookAcommodation should occur with a price lower than 15.

Template name	**LTL semantics**	**Description**
Init(A)	A	Task A should be first.
Existence(A, 1)	F A	Task A should occur at least once
Absence(A, 1)	¬ Existence(A, 1)	Task A should never occur
Existence(A,N)	F(A ∧ X(Existence A, N - 1))	Task A should occur at least N number of times
Absence(A, N)	¬ Existence(A, N)	Task A can occur N-1 times at most
Exactly(A,N)	Existence(A, N) ∧ Absence(A, N+1)	Task A should occur exactly N times
Responded existence(A,B)	G(A ⇒ (OB ∨ FB))	If task A occurs than task B must also occur
Choice(A,B)	FA ∨ FB	Either task A or B or both should occur
Exclusive choice(A,B)	(FA ∨ FB) ∧ ¬ (FA ∧ FB)	Either task A or B should occur but not both
Response(A,B)	G(A ⇒ FB)	If task A occurs then task B occur after task A
Chain response(A,B)	G(A ⇒ XB)	If task A occurs then task B must occur next
Alternate response(A,B)	G(A ⇒ X(¬ A U B))	If task A occurs then task B must occur also, without another task A between those two tasks
Precedence(A,B)	G(B ⇒ OA)	If task B occurs than task A should occur before task B
Alternate precedence(A,B)	G(B ⇒ Y(¬ BSA))	If task B occurs then task A should also occur before B, without having another task B between those tasks
Chain precedence(A,B)	G(B ⇒ YA)	If task B occurs then the previous task should be task A
Not response(A,B)	G(A ⇒ ¬ FB)	If task A occurs then task B should not occur after task A
Not chain response(A,B)	G(A ⇒ ¬ XB)	If task A occurs then task B should not occur next
Not responded existence(A,B)	G(A ⇒ ¬ (OB ∨ FB))	If task A occurs then task B should not occur
Not precedence(A,B)	G(B ⇒ ¬ OA)	If task B occurs then task A should not occur before task B
Not chain precedence(A,B)	G(B ⇒ ¬ YA)	If task B occurs than the previous task should not be A

Table 3. Declare constraints

2.5 Alloy

Alloy [13] is an encoding for model checking with SAT. In Alloy, there is a Signature (sig) declaration that is equivalent to a class in object-oriented programming languages. Keyword “fact” is used to constrain the model. The constraints in the fact block are assumed to be always true. A function is defined with keyword “fun” and it has parameters that Alloy uses to do computations and return some value. A predicate (pred) is a function that returns a boolean value.

Alloy analyzer is an open source model checker that is able to check the model correctness. It can produce valid examples which conforms to the given model constraints and also produce examples which violate the given restrictions. Alloy analyzer takes an input model that is given in the form of Alloy code and uses the boolean SAT solver [14] to find a solution. It can find all the possible models which conform to the given boolean formula.

3 Related work

This section covers the existing state-of-the-art research papers concerning conformance checking with Declare, MP-Declare and other data-aware approaches with imperative modeling languages.

3.1 Conformance checking with imperative models without data

There are many papers concerning imperative models focusing on control flow without data. These works replay the event log on the model and then measure conformance by comparing the stream of events that is that the model generates and the stream of events that is gotten from the log[15, 16, 17]. In alignment based techniques, compliance is checked by taking the modelled behavior and the actions seen in the event log and then aligning them both against each other [18].

3.2 Conformance checking with Declare

In [19], an alignment-based approach is used for conformance checking using Declare models. An alignment between a model and a log means that the log transitions from one event to another in the log are related to the transitions of one activity to another activity in the model. There might be instances where some transitions in the log cannot be imitated by the model and this might be also the other way around. There exist three categories of moves after an alignment:

- (e , ≫) is a transition in the log where there is an event e from the log that could not be related to a move in the model.
- (≫, a) is a transition in the model where in the model there is an activity a that needs to be executed, but there was no related event occurrence in the log.
- (e,a) is a synchronous transition where event e occurred and it corresponds to an activity a executed in the model.

The more synchronous move the better and the other types of moves should ideally not occur at all. The fitness of a trace is computed based on the alignment of a trace with respect to the model. Fitness is a numerical value between 0-1. If fitness is 1 then the alignment has only synchronous moves.

In [12] an approach is described that is used for evaluating the compliance of a log w.r.t. a Declare model. For each trace it shows whether a constraint is classified as satisfied or violated and also the healthiness of a log is reported. In this approach the Declare constraints are converted into automata and an activation tree is used to identify whether the trace is violated or satisfied. This approach only checks for control-flow violations.

3.3 Conformance checking with data-aware imperative models

An alignment-based approach is also used in [20] and in [21]. Both use Petri nets but the latter approach is also Resource-Aware.

3.4 Multi-Perspective Declare conformance checking

In [3], an approach for MP Declare conformance checking is presented. An implementation is provided as a ProM plugin.

In [22], an approach is presented for conformance checking with respect to constraints where the control-flow is expressed in Declare, but the data perspective is expressed in terms of conditions on global variables. These variables are not connected to the Declare constraints and so the behavior is not tied to the data attributes.

3.5 Compliance Monitoring

One of the earliest tools for runtime monitoring using Declare and implemented in ProM is [23]. With this tool, it is possible to check individual control flow constraints and also to detect conflicts between two or more constraints. MoBuCon EC [24] is a tool where along with the control flow it is also possible to monitor time related constraints at runtime. It does not have the feature to detect conflicts. In [25], a ProM plugin is introduced where Integer Linear Programming (ILP) is used at runtime to monitor data and time aware declarative models and it also enables early violation detection. However, this approach does not handle conflicts between more than two constraints. In Figure 1, a comparison of different runtime monitoring tools is given [2].

APPROACH	CMF 1 time	CMF 2 data	CMF 3 resources	CMF 4 non-atomic	CMF 5 lifecycle	CMF 6 multi-instance	CMF 7 reactive mgmt	CMF 8 pro-active mgmt	CMF 9 root cause	CMF 10 compl. degree
Superv. Control Theory [17]	+/−	−	+	+	+	−	−	+	−	−
ECE Rules [31]	+	+/−	+	+	−	−	+	−	+/−	+
BPath (Sebahi) [42]	+	+	+	+	+/−	+	+	−	−	+/−
Gomez et al. [34]	+	−	−	+	n.a.	+/−	+	+	−	−
Giblin et al. [33]	+	n.a.	n.a.	n.a.	n.a.	n.a.	+	n.a.	n.a.	n.a.
Narendra et al. [40]	−	+	+	n.a.	−	+	+	−	−	+
Thullner et al. [41]	+	n.a.	n.a.	n.a.	n.a.	n.a.	+	−	−	n.a.
MONPOLY [26,27]	+	+	+	+/−	+/−	+	+	−	−	−
Halle et al. [24]	+/−	+	+/−	n.a.	n.a.	n.a.	+	n.a.	n.a.	n.a.
Dynamo [21–23]	+	+	+/−	+	n.a.	+	+	−	−	+/−
Namiri et al. [18]	+/−	+	+	+	−	+	+	−	−	−
MobuconEC [39]	+	+	+	+	+	+	+	−	−	+/−
Mobucon LTL [36–38]	+/−	−	−	+	−	−	+	+	+	+/−
SeaFlows [35]	+/−	+/−	+/−	+	+	+	+	+	+	+/−

Caption: + supported, + implementation publicly available, +/− partly supported, − not supported, n.a. cannot be assessed.

Figure 1. Compliance monitoring approaches [2].

4 Problem

Compliance monitoring is an important functionality for a process-aware information system (PAIS). Compliance monitoring is the ability to check at runtime whether what can be observed from the logs is compliant with a given business process model. It was developed in order to prove at runtime that business processes are compliant with regulations, laws and guidelines. In this thesis, the topic is checking compliance at runtime with respect to MP Declare which means monitoring the progress of process instances to detect violations and conflicts with respect to MP-Declare constraints.

In this thesis, the model checker Alloy Analyzer is used to check whether a process execution is compliant with respect to MP-Declare constraints. In particular, the research questions that this thesis tries to answer are:

1. How to use Alloy to monitor individual data-aware constraints?
2. How to use Alloy to detect conflicts due to the interplay of data-aware constraints?
3. Is this approach applicable to real-life case studies?

To answer the first question an explanation on how constraints and traces will be encoded into Alloy is given. Starting from this encoding we show how the Alloy Analyzer can be used to express the state of individual constraints using four valued runtime verification semantics. The answer to the second question will be similar to the first one but there will be some extra steps to allow the Alloy Analyzer to detect conflicts between two or more constraints. To answer the third question, the approach will be applied to a real-life event log pertaining to the process of treating patients in a Dutch hospital.

5 Approach

In this section an overview of the proposed approach is given. This approach will allow us to answer the research questions stated in the previous Section.

5.1 Model representation in textual form

A MP-Declare model can be represented either graphically or textually. In this paper, modeling will be done textually using the approach developed in [7]. This textually represented model will be used to generate an Alloy model that is used as an input for the Alloy Analyzer.
There are 6 different categories of statement types which can be used in the textual model representation which are: activity, constraint, data constraint, trace attribute, data attribute and the bind between activity and data.
Activities have a name that is a string value. An example of how activities are represented in this format is:

```
activity ApplyForTrip
activity BookTransport
activity BookAccommodation
activity CollectTickets
```

There are three data types which can be used: integer, float and enumerative. An example of how integer and float can be used is:

```
Price: float between 0.5 and 235.5
Age: integer between 0 and 130
```

An example of enumerative data is:

```
TransportType: Bus, Car, Train, Plane
```

Activity and data binds are represented as:

```
bind BookAccommodation: Price
bind BookTransport: TransportType
bind CollectTickets: TransportType
bind BookTransport: Price
```

There are three types of constraints: unary, unary with index and binary constraints. Unary constraints are constraints that have only one activity for example *existence(A)*, unary constraints with index are constraints that in addition to the activity name have an integer parameter (an index) for example *absence(A, N)*. Binary constraints are constraints where along with the activation activity there is also a target activity like *response(A,B)*. Some examples of how to express constraints in textual form are:

```
Init[ApplyForTrip]
Precedence[CollectTickets, BookAccommodation]
Exsistence[CollectTickets]
Absence[ApplyForTrip, 2]
```

Data constraints have a function after the constraint declaration. Data constraints also may have a variable name after the activity name:

```
Absence[BookTransport A] | A.Price>30
```

In this example, the activity name *BookTransport* is assigned with variable *A* and using the latter we define function $A.Price > 30$. This means that events with activity name *BookTransport* where the *Price* is higher than 30 should not occur. Comparison operators that can be used with data constraints where data is of type integer or float are: >, <, >=, <= and =.
When using enumerative data, keywords that can be used are: *is*, *is not*, *in*, *not in*, *same* and *different*. Examples of how to use these keywords are:

```
Exsistence[BookTransport] | A.TransportType in (Train, Bus)
Exsistence[BookTransport] | A.TransportType not in (Plane, Car)
Response[BookTransport, CollectTickets] | same TransportType
Response[BookTransport, BookTransport] | A.TransportType is Bus |
        T.TransportType is not Car
```

The first constraint says that there must be an activity *BookTransport* with 'TransportType' from the set (Train, Bus) and the second one says that there must be an activity *BookTransport* with 'TransportType' is not from the set (Plane, Car). The third one says that *BookTransport* activities should be followed by *CollectTickets* with the same 'TransportType'. If keyword *different* is used instead of *same* that would mean that the events must have different 'TransportType'. The second to last data constraint says there cannot be an activity *BookTransport* where Price is smaller than 100 or a 'TransportType' with a value Plane. The last data constraint from this example says that *BookTransport* with a 'TransportType' that has value Bus has to be followed somewhere in the trace by an activity *BookTransport* with 'TransportType' that is something other than Car.

It is also possible to use operators on arguments which are prioritized in this order: *not*, *and*, *or*:

```
Exsistence[BookTransport]| A.Price<100 and A.TransportType is Plane
Absence[BookTransport]| A.Price<100 or A.TransportType is Plane
Exsistence[CollectTickets]| not A.TransportType is Car
```

The second data constraint says there cannot be an activity *BookTransport* with Price smaller than 100 or 'TransportType'equal to Plane.

5.2 Encoding in Alloy

The Alloy encoding of an MP-Declare model is presented in [7]. Here, a summary of that approach is given.

5.2.1 Encoding MP-Declare into Alloy

In Alloy, objects are called signatures and the keyword for it is *sig* and they can be abstract. In this case it is needed to define a signature Activity like this:

```
abstract sig Activity{}
```

It is also possible to define child objects that extend that abstract signature that was previously defined. Activity *CollectTickets* would be defined like this:

```
one sig CollectTickets extends Activity{}
```

Traces are made up of events. Each event has to have a reference to an activity and for this another abstract signature is defined with a reference to an activity.

```
abstract sig Event{
        task: one Activity
}
```

An instance of an event would be defined like this:

```
one sig TE0 extends Event{}
```

where "TE0" is the name of the event. In this implementation the trace will always have a fixed size. If the fixed size of the trace is 3, then we define "TE0", "TE1" and "TE2" that all extend the abstract signature of Event. It is important to check the ordering between two events to see if one event comes before or after another event for constraints like *response* and *precedence*. It is also important to check whether one event is right next to another event for constraints such as *ChainResponse* and *ChainPrecedence*. Because of this it is necessary to define two predicates in Alloy code called "next" and "after":

```
pred Next(p, n: Event){p=TE0 and n=TE1 or p=TE1 and n=TE2
or p=TE2 and n=TE3 or p=TE3 and n=TE4 or p=TE4 and n=TE5}
```

This predicate has two arguments where "p" stands for "previous" and "n" stands for "next".

```
pred After(b, a: Event){b=TE0 and not (a=TE0) or b=TE1 and not
  (a=TE1 or a=TE0) or b=TE2 and not (a=TE2 or a=TE0 or a=TE1)
  or b=TE3 and (a=TE5 or a=TE4) or b=TE4 and (a=TE5)}
```

Here, the argument “b” stands for “before” and “a” stands for “after”. A predicate in Alloy is a function that returns a Boolean value. These predicates are generated for a trace where maximum length is 6 but it is possible to have them either smaller or larger.

Constraints are also encoded using predicates. Here, we give some examples of how some of the Declare constraints are encoded in Alloy. *Init(A)* means that activity *A* has to be the first one to occur in a trace and in Alloy this is encoded as:

```
pred Init(taskA: Activity) {taskA = TE0.task}
```

From this example it is possible to see that we can specify the exact position of an event in the trace as a constraint. *Absence(A)* predicate would be represented as:

```
pred Absence(taskA: Activity) {no te: Event | te.task = taskA}
```

Here “te” means “task event”. This can be interpreted as: “There is no task event with task name equal to taskA”. *Exactly(A,N)*:

```
pred Exactly(taskA: Activity, n: Int) {#{ te: Event | taskA = te.task } = n}
```

Here the cardinality operator # is used to count the task events. This means that the amount of *Events* where task name is equal to *taskA* is *N* times. *Chain response(A,B)*:

```
pred ChainResponse(taskA, taskB: Activity) {
all te: Event | taskA = te.task implies
(some fte: Event | taskB = fte.task and Next[te, fte])
}
```

This can be read as: “For all the task events where task name is *taskA*, there has to be another task event where task name is *taskB* and the latter must be next to *taskA*”.

Everything in a *fact* block in Alloy will always hold true when the Alloy Analyser generates a solution. Therefore, when all the constraints in the Multi-Perspective Declare model are expressed as predicates in Alloy then the constraints of the model are put all into a *fact* block:

```
fact{
    Response[ApplyForTrip, BookTransport]
    Init[ApplyForTrip]
    Exactly[ApplyForTrip,3]
    Absence[CollectTicket]
}
```

In order to have data attributes, then a generic abstract signature “Payload” has to be defined to encode the data into Alloy.

```
abstract sig Payload{}
```

For example, we can define a "TransportType" where the possible values are Bus and Plane. The Signature for this attribute is abstract while the values are not as they are the subclasses of "TransportType".

```
abstract sig TransportType extends Payload {}
one sig Bus extends TransportType{}
one sig Plane extends TransportType{}
```

We also need to add a payload to every Event:

```
abstract sig Event{
        task: one Activity,
        data: set Payload
}
```

The keyword *set* means that for one event there can be 0 to many data attributes. If, for example, we would like to tie the TransportType with the activity *CollectTickets* then the Alloy code would look like:

```
fact{all te: Event | (lone TransportType & te.data)
all te: Event | some (TransportType & te.data) implies te.task in (CollectTickets)}
```

The keyword "lone" means 0 to 1. The first line can be read as for each event there can be 0 to 1 value of TransportType. So there cannot be an occurrence where an activity would have more than one TransportType value. The second line means that the TransportType is tied to *CollectTickets*.

When data constraints are encoded in Alloy, they are regular constraints with an extra function for data. For example, take the following data constraint:

```
        Response[BookTransport, BookTransport] | A.TransportType is Bus |
T.TransportType is not Train
```

In order to express activation and correlation conditions in Alloy we need to define two predicates:

```
pred p1a(A: Event) { { (A.data&TransportType=Bus) } }
pred p1b(T: Event) { { (not T.data&TransportType=Train) } }
```

The first predicate returns true if the TransportType data attached to the activation event is equal to *Bus*. The second predicate says that the TransportType data attached to the target event should not be equal to *Train*. In Alloy code the response constraint above would be expressed as:

```
fact{all te: Event |(BookTransport = te.task and p1a[te]) implies
(some fte: Event |BookTransport = fte.task and p1b[fte] and After[te, fte])}
```

If the data constraint was:

```
Response[BookTransport, CollectTickets] | same TransportType
```

then the predicate would be:

```
pred p2a(A, B: Event) {{(A.data&TransportType = B.data&TransportType)}}
```

A problem with Alloy is that it does not have support for numeric values. When numbers are involved they are mapped to some predefined intervals and presented as enumerated values. For example, if there is a Price field and a constraint:

```
Price: float between 0 and 100
Absence[BookTransport] | A.Price>30
```

with the first line we declare that there is a field Price which can range from 0 to 100 and then with the constraint we say that there should not be any activity *BookTransport* where Price is higher than 30. As mentioned previously Alloy does not support numbers so they are represented as intervals and with this example 2 intervals will be created which are: float between 0 and 30, float between 30 and 100 which will be encoded into Alloy like this:

```
one sig floatBetween30p0and100p0 extends Price{}
pred p(A: Event) {{A.data&Price in (floatBetween30p0and100p0)}}
no te: Event | te.task = BookTransport and p[te]
```

30p0 means 30.0 and 100p0 means 100.0 (because Price is a float).

5.2.2 Encoding traces in Alloy

We can assign activity and data to some or all the events in a trace. We can also set the exact position of an event in the trace. The trace given in Listing 1 would be represented in Alloy as:

```
fact{
    ApplyForTrip = TE0.task
    BookTransport = TE1.task
    Car = TE1.data & TransportType
    floatBetween30p0and100p0 = TE1.data & Price
    BookAccomodation = TE2.task
    floatBetween0p0and30p0 = TE2.data & Price
}
```

5.3 Run time constraint monitoring

Here, we describe how constraints are monitored in an on-line setting. The first input is an MP-Declare model in textual form as described in Section 5.1. At runtime, a complete log is not available. Instead of that, single events are coming and are processed one by one. Traces are partial, ongoing traces.

5.3.1 Monitoring individual constraints

When an event comes in, it is translated into Alloy code as described in Section 5.2.1. Then, the tool iterates through the constraint list and checks each constraint individually. In one iteration, the tool generates a .als file which describes the constraint that is being checked and the partial trace in Alloy code and runs it in the Alloy Analyzer. The latter generates the conjunctive normal form of the model and then solves it using a SAT solver [14]. Once solved the Alloy analyzer returns a Boolean value specifying whether the SAT solver was able to find a solution or not. In this implementation, Alloy needs a fixed size which means that the maximum trace length should be specified. For different constraints, the maximum trace length that must be specified is different. With some constraints such as response and responded existence, the maximum trace size is the size of the partial trace increased by the number of pending activations of that particular constraint in the partial trace. Increased trace sizes for different constraints are shown in Table 4.

Constraint	**Increased Trace size**
Response(A,B)	number of pending activations
RespondedExistence(A,B)	number of pending activations
ChainResponse(A,B)	1
AlternateResponse(A,B)	1
Existence(A)	1
Existence(A,N)	*N*
Exactly(A,N)	*N*
ExclusiveChoice(A,B)	1

Table 4. Increased trace size w.r.t. the constraint

When we have determined if the SAT solver could find a solution or not, a state which is based on a four-valued semantics is assigned to the partial trace accordingly. The values of the four-valued semantics [23] used to describe the state of a constraint are:

- **Possibly satisfied** - the constraint is presently satisfied, but there is the possibility of it being violated in the future.

- **Possibly violated** - the constraint is presently violated, but it might be satisfied in the future.
- **Permanently satisfied** - the constraint is satisfied can not be violated anymore.
- **Permanently violated** - the constraint is violated can not be satisfied anymore.

While monitoring traces with respect to a constraint, one of the semantic values for the current state is given to the constraint out of the four values previously mentioned. The value of the state depends on the type of constraint. Table 5 shows the criterion upon which the value of the state will be assigned to a constraint. In Table 5 the following abbreviations are used:

- **Violations** v - number of violations in the current prefix.
- **fulfillments** f - number of fulfillments in the current prefix.
- **pending activations** p - number of pending activation in the current prefix.
- **done** - current trace has received the last event.

In Table 5 *existence(A, N)* is violated when $done \wedge (a < N)$. This means that in order for the constraint to have the value violated the current trace has to be finished and the total number of activations has to be smaller than N which is the number of times activity A has to occur.

Algorithm 1 shows the steps that are made to assign a state to a *Exclusive choice* constraint. Once all the events are completed the log streamer sends in activity "completed" and then each constraint is assigned a final state.

5.3.2 Conflict detection

Section 5.3 dealt with monitoring of individual constraints. However, in a model where there are more than one constraint it is not enough to monitor individual constraints because it is desirable to know whether two or more constraints contradict each other. When constraints are in conflict with each other then at least one of them will be violated when the trace completes. Because of this a new semantic value is introduced in order to show which constraints are in conflict. Like in the case of individual constraints, also when monitoring the conjunction of two or more constraints for conflict detection, an upper bound for the size of the possible continuations of the partial trace needs to be specified. However, differently from the case of individual constraints, when monitoring a conjunction of constraints, computing this upper bound is not trivial. Nevertheless, if we assume to have some background knowledge about the maximum length of the possible traces of a process, e.g., derived from a process model or from an event log, we

Template	Possibly satisfied	Possibly violated	Violated	Satisfied
init(A)	–	–	$v > 0$	$f > 0$
absence(A) absence(A,N) precedence(A,B) alternate precedence(A, B) chain precedence(A, B) not precedence(A, B) not chain precedence(A,B) not response(A,B) not chain response(A, B) not responded existence(A,B)	–	$!done \wedge (v = 0)$	$v > 0$	$done \wedge (v = 0)$
response(A) responded existence(A,B)	$!done \wedge (p > 0)$	$!done \wedge (p = 0)$	$done \wedge (p > 0)$	$done \wedge (p = 0)$
existence(A) existence(A,N)	$!done \wedge (a < N)$	-	$done \wedge (a < N)$	$a >= N$
exactly(A,N)	$!done \wedge (a < N)$	$!done \wedge (a = N)$	$(a > N) \vee (done \wedge (a < N))$	$done \wedge (a = N)$
chain response(A,B) alternate response(A,B)	$!done \wedge (v = 0) \wedge (p > 0)$	$!done \wedge (v = 0) \wedge (p = 0)$	$(v > 0) \vee (done \wedge (p > 0))$	$done \wedge (v = 0) \wedge (p = 0)$
choice(A, B)	$!done \wedge (a = 0)$	–	$done \wedge (a = 0)$	$a > 0$
exclusive choice(A, B)	$!done \wedge (a = 0)$	$!done \wedge (v = 0) \wedge (a > 0)$	$(v > 0) \vee (done \wedge (a = 0))$	$done \wedge (v = 0) \wedge (a > 0)$

Table 5. Basis for semantic values

Algorithm 1: ExclusiveChoice constraint checking

Input: Constraint, partial trace
Result: Setting the state of the constraint

```
1  maxTraceLength = partialTrace.size;
2  hasSolution = AlloyCheck(Constraint, maxTraceLength);
3  if hasSolution then
4  |   state = possiblySatisfied;
5  else
6  |   maxTraceLength = maxTraceLength +1;
7  |   hasSolution = alloyCheck(Constraint, maxTraceLength);
8  |   if hasSolution then
9  |   |   state = possiblyViolated;
10 |   else
11 |   |   state = permanentlyViolated
```

can use this knowledge to determine this upper bound. For example, in our experiments, the upper bound is set to the biggest trace size in the event logs we use.

Conflict detection with a Simple case. Consider two constraints *Response(A,B)* and *Absence(B)*. The first rule says that when activity *A* occurs, then afterwards activity *B*

must also occur. The second rule says that there cannot be an activity *B*. As soon as activity *A* occurs then one of these constraints will be violated. If *B* occurs, then the absence constraint will be violated and if it does not occur then the response constraint will be violated. This means that these two constraints are in conflict with each other when *A* occurs.

Table 6 shows the same constraints with data added to them. The constraints are the same as in the first example but since data attribute x was added to them they are no longer in conflict. If we take the response constraint, then this constraint can be activated only when activity *A* occurs with data x = 10. This constraint has also a target condition which says that an activity *B* with x = 3 has to occur. The *absence* constraint says that activity *B* with x = 4 cannot occur. When activity *A* occurs then these constraints are not in conflict anymore and these conditions can be satisfied at the same time.

Id	**Constraint**	**Activation Activity**	**Target Activity**	**Activation Condition**	**Target Condition**
1	*Response*	*A*	*B*	*A.x = 10*	*T.x = 3*
2	*Absence*	*B*	-	*A.x = 4*	-

Table 6. Non conflicting constraints with data

In Table 7, activation and correlation conditions are changed. Now the correlation condition of constraint *Response(A, B)* is the same as the activation condition of *Absence(B)*. When the response constraint is activated it requires that activity *B* with data x = 4 to come in, but the *absence* constraint does not allow activity *B* with data x = 4 to occur. So these two constraints are in conflict as soon as the response constraint is activated.

Id	**Constraint**	**Activation Activity**	**Target Activity**	**Activation Condition**	**Target Condition**
1	*Response*	*A*	*B*	*A.x = 10*	*T.x = 4*
2	*Absence*	*B*	-	*A.x = 4*	-

Table 7. Conflicting constraints with data

In order to detect conflicts with Alloy, we take the same steps as mentioned in Section 5.3.1 where we monitor individual constraints. But instead of translating only one constraint into Alloy, both of them are translated.If the Alloy Analyzer can find a solution, then it means that the checked constraints are not in conflict. If it cannot find a solution, then the checked constraints are in conflict and both of these constraints will be given a semantic value of “conflict”.

Conflict detection with indirect conflicts. In the previous section, it was discussed how using Alloy it is possible to find conflicts between two constraints. However, when the number of constraints is greater, the conflict can be caused by more than two constraints and the method for detecting conflicts becomes more complex. Consider the three constraints without data:

1. *Response (A,B)*
2. *Response (B,C)*
3. *Absence(C)*

In this example, when activity *A* occurs, rule 1 is activated and activity *B* must occur and when it occurs then rule 2 is activated and activity *C* must also occur in order for the rule to be satisfied. However, rule 3 says that activity *C* cannot occur. This means that when *A* occurs it will cause a conflict between all these constraints.

In order to find this type of conflict, again, all the constraints with the partial trace are checked in conjunction with the Alloy Analyzer. If the Alloy Analyzer cannot find a solution, then there is a conflict. The problem is that, when the constraint set contains more than 2 constraints, it is not known which subset of constraints is causing the conflict. In order to find out the minimum set of conflicting constraints all the combination of constraints are tried out. In this case, we would first have pairs of rules: 1, 2; 2, 3 and 1, 3. Now all of these constraint pairs along with partial trace are checked again using Alloy Analyzer. However, this would not find any conflict because there is no conflict between any of those pairs. So we increase the size of the sublist and make combinations again. This example only has three constraints so there is only one combination. With this example the Alloy Analyzer cannot find a solution and these constraints will be assigned a state value of “conflict”. Algorithm 2 describes this process.

Algorithm 2: Finding conflicting constraints

Input: Set of Constraints
Result: Setting the state for the conflicted constraints

```
1 sizeOfSublist = 2;
2 while solution has been found do
3     sublists = makeSubLists(Constraints, sizeOfSubList);
4     foreach sublist in sublists do
5         hasSolution = alloyCheckforConflict(sublist)
6         if hasSolution == false then
7             foreach constraint in sublist do
8                 constraint.state = conflict;
9     sizeOfSublist = sizeOfSublist +1;
```

6 Implementation

This technique was developed to be integrated into a tool for rule mining [4], developed in Java 8[1]. The tool with the extension is available at GitHub: `https://github.com/b26140/Rule-mining-tool-with-monitor-extension`. The source code of the extension that was integrated into the tool can be found on the public bitbucket repository[2]. In Figure 2, the front panel of the application is shown. This tool also provides an option to communicate with an Information System at runtime and specify the port from where sending the model and stream of events will be received. When inserting a model in the front panel, it will give the user an overview of activities, constraints, data and activity to data binds contained in the model. The tool also gives an option to run it without conflict check.

Instead of using a real Process-Aware information system connected with the tool, a log streamer is used to simulate a process execution. Log streamer is there to send a model and then a stream of events to the core module through a socket. This log streamer is integrated into the application.

The architecture of the application is depicted on Figure 3. Through the GUI module the user can submit a log and a model. The model and the log are passed to the Log Streamer. The first task of the log streamer is to send the model to the core module that parses it. After that the Log Streamer starts sending the stream of events to the core module, the events are also parsed and an .als file is created which is an Alloy code file that represents the model and the partial trace. The Core module then asks the Alloy Analyzer to analyze the .als file. The analyzer reads the .als file, solves it and then sends

[1] https://docs.oracle.com/javase/8/

[2] https://bitbucket.org/KKKristjan/declareconformancechecker/src/master/

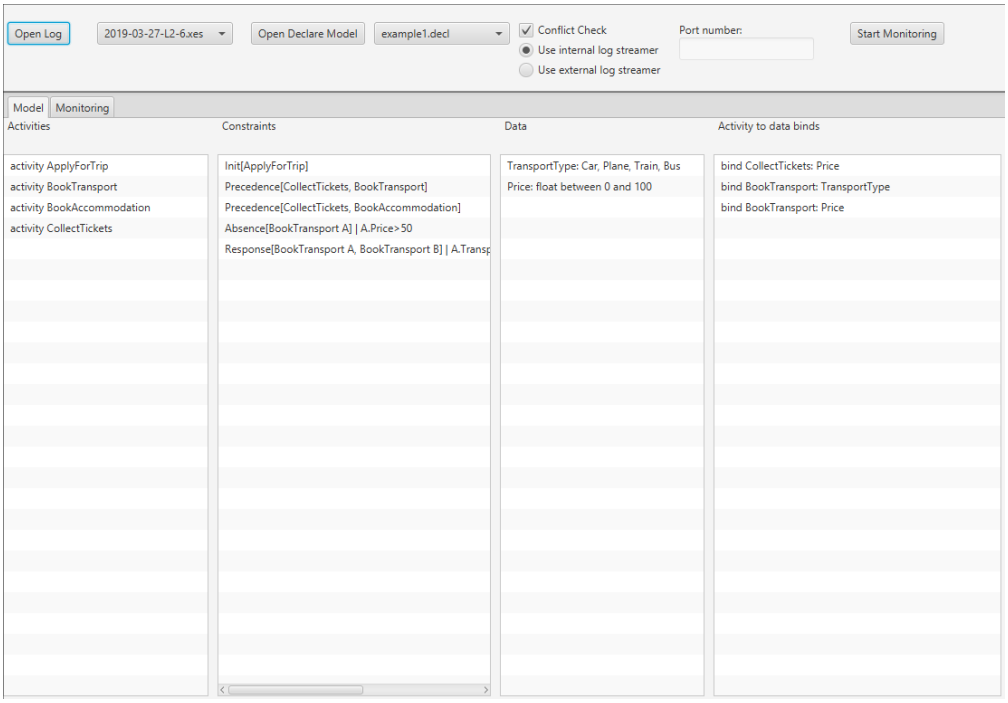


Figure 2. Front panel of the application

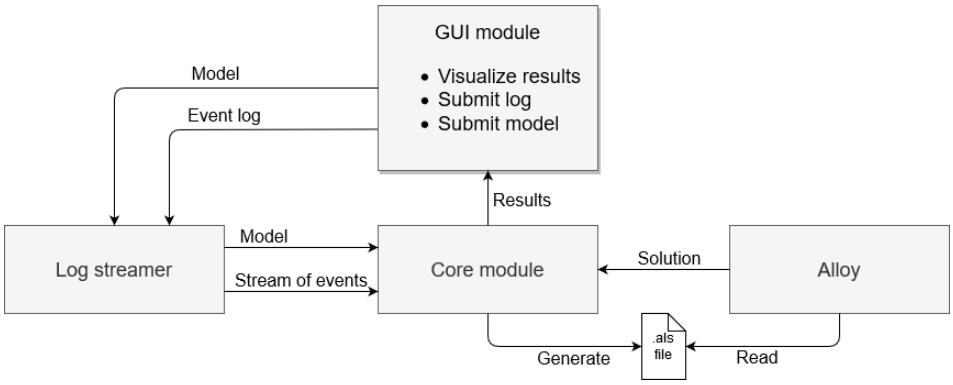


Figure 3. Architecture of the implementation

back whether it was able to find a solution or not. Based on the Alloy Analyzer outcome, the core module does some further analysis and then sends a response to the GUI module which then visualizes the results for the user as shown in Figure 4:

1 **Case selector** - Enables the user to select a case to view the results visualization.

2 **Constraints with data conditions** - Shows the constraint name and the activities and data conditions related to that constraint.

3 **Event details** - Shows the details related to an event such as activity name, data and timestamp.

4 **State of the constraint** - Shows the state of a constraint in color-coded format. Each color matches with a different semantic value.

The application uses the same visualization techiniques as the MoBuConLTL [2].

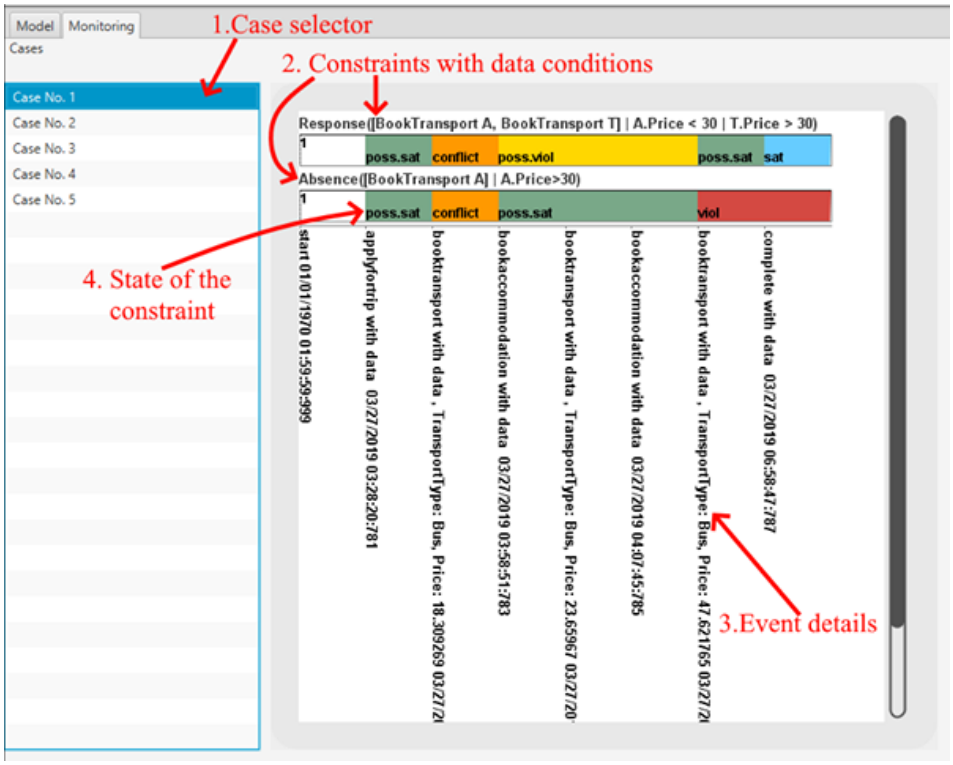


Figure 4. Monitor GUI

[2]https://svn.win.tue.nl/repos/prom/Packages/MoBuConLTL/Trunk

7 Evaluation

We evaluate our solution in two different ways. First, we look at the correctness of the output. Then, we test the performance of the monitor. In order to verify the correctness of the output, we use different sets of event logs generated by [7] and different business rules. First, we use simple artificial examples and then we increase their complexity. In addition, a real life event log is used in order to measure the performance of the approach in a real-life scenario. Finally, the correctness of the solution will be compared to another state-of-the-art solution.

7.1 Example 1

Firstly, the proposed approach was tested on a single constraint. Table 8 describes the individual constraint with activation and target conditions that was monitored. Figure 5shows the output of the monitor for three different traces. In trace 1, the activity *Book transport* with *Type* equal to *Bus* never occurs, therefore, the constraint is never activated. The constraint is always possibly satisfied, becoming permanently satisfied when the trace completes. In trace 2, the activation condition is met when the second event occurs and the constraint changes state from possibly satisfied to possibly violated. Later during the execution, activity *Collect Tickets* with *T.Price > 30* occurs making the constraint to change its state back to possibly satisfied. In trace 3, the constraint is activated the target does not occur which makes the constraint violated when the trace completes.

Id	**Constraint**	**Activation Activity**	**Target Activity**	**Activation Condition**	**Target Condition**
1	*Response*	*Book transport*	*Collect tickets*	*A.Type is bus*	*T.Price < 30*

Table 8. Model with a single constraint

(a) Trace 1 (b) Trace 2 (c) Trace 3

Figure 5. Output of the first example

7.2 Example 2

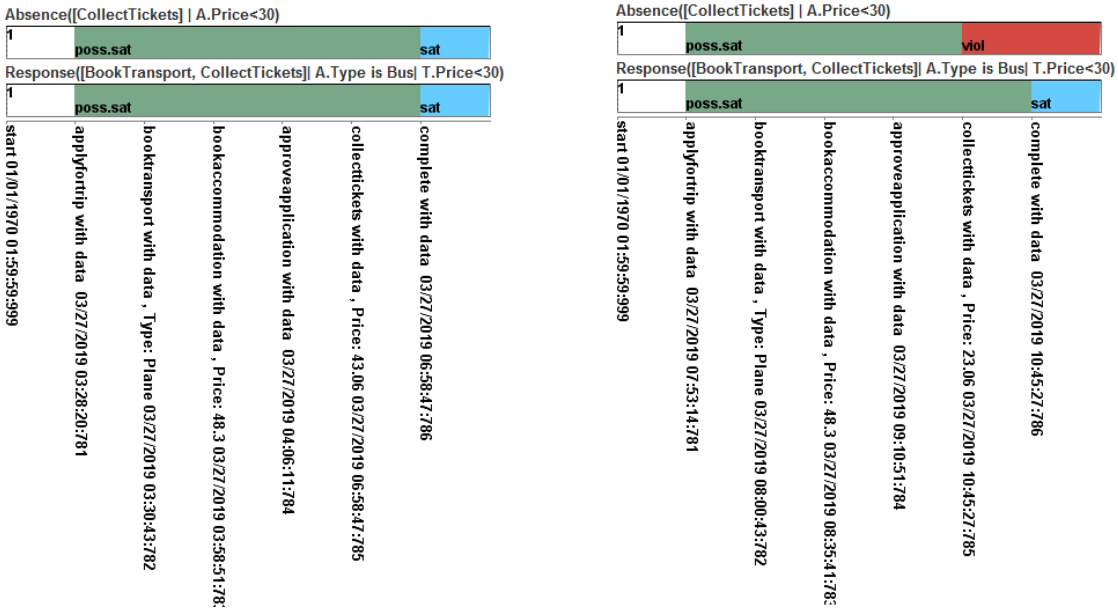
We now use the two business constraints described in Table 9. The response rule says that when activity *Book transport* with *Type* equal to *Bus* occurs, then *Collect tickets* with *Price<30* must occur afterwards. The absence rule says that activity *Collect tickets* cannot occur with a price lower than 30.

Id	Constraint	Activation Activity	Target Activity	Activation Condition	Target Condition
1	*Absence*	*Collect tickets*	-	*A.Price < 30*	-
2	*Response*	*Book transport*	*Collect tickets*	*A.Type is bus*	*T.Price < 30*

Table 9. Example 2 constraints

In Figure 6, the output of the monitor for two different traces is shown. In trace 1, the two constraints are never activated and become permanently satisfied when the trace completes. In trace 2, rule 1 is not activated. However, an event *Collect tickets* with *Price* 23,06 occurs so that rule 2 becomes permanently violated.

In Figure 7, the output of the monitor for a third trace is shown without and with conflict detection. The second event of the trace is *Book transport* with *Type* equal to *Bus*, which means that rule 2 is activated. However, since the target activity with a valid target condition never occurs, the constraint is violated. Notice that when Book transport activity occurs and conflict detection is turned on the Alloy model checker can see that there is no finite continuation of the trace that does not violate at least one of these two constraints, which means that they are in conflict with each other as shown in the Figure 7.



(a) Trace 1 (b) Trace 2

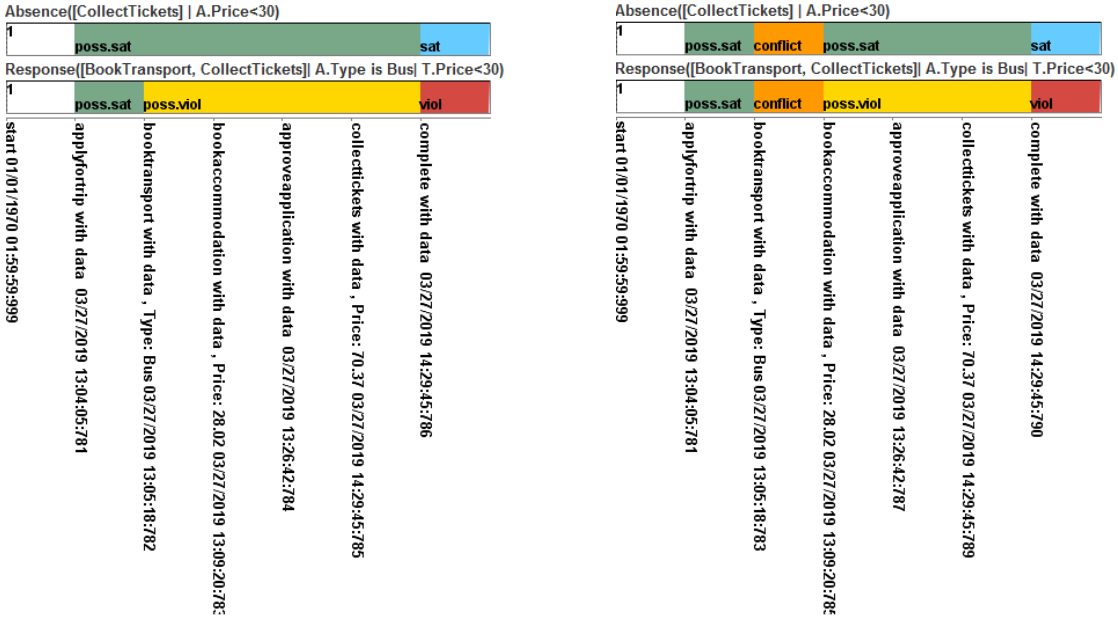
Figure 6. Output of the monitor for Example 2 and Traces 1 and 2

7.3 Example 3

The constraints for this example are shown in Table 10. This example will also showcase that the monitoring tool is able to detect conflicts between two business constraints that are directly related to one another. It is possible with this tool to have a binary constraint in the model where there might be a function specified for one activity name but not the other. In this example the two constraints have the same target activity. While the previous example showed that the tool is able to find conflicts between an unary and a binary constraint then this examples shows conflict detection between two binary constraints.

As can be seen in Figure 8,in trace 1, rule 2 is activated. However, when activity *Book accommodation* occurs, rule 1 is not activated because the activation condition $A.Price < 30$ is not satisfied (price is 43,06). This means that activity *Approve application* cannot occur. Therefore, the two constraints are now in conflict. which means that one of the constraints will be violated. Figure 8 shows that rule 1 is eventually violated.

This example demonstrates that the tool was able to detect a possible violation because of activity *Approve application* as soon as both of the constraints were activated.



(a) Without conflict detection

(b) With conflict detection

Figure 7. Output of the monitor for Example 2 and Trace 3

Id	Constraint	Activation Activity	Target Activity	Activation Condition	Target Condition
1	*NotResponse*	*Book accommodation*	*Approve application*	*A.Price < 30*	-
2	*Response*	*Book transport*	*Approve application*	*A.Type is bus*	-

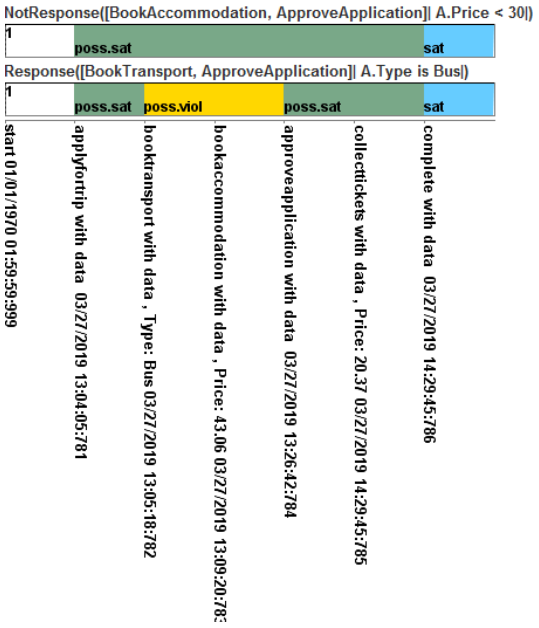
Table 10. Example 3 constraints

7.4 Example 4

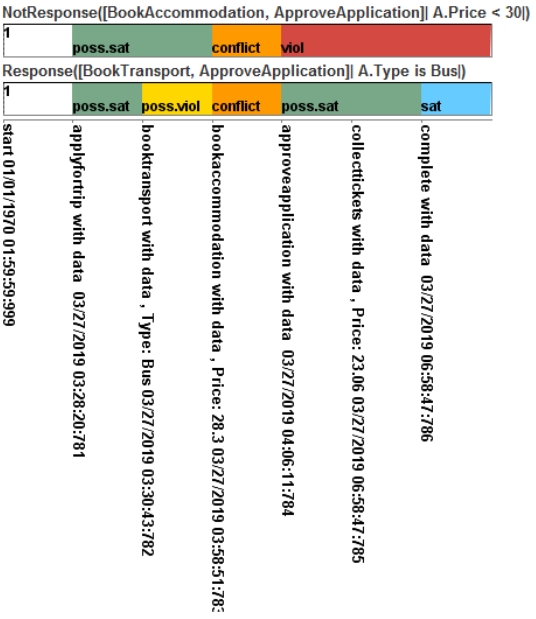
The constraints we use in this example are shown in Table 11. In the previous two examples it was shown that the tool is able to detect conflicts between two constraints. This example was chosen in order to showcase that the tool is able detect conflicts caused by three constraints.

Id	Constraint	Activation Activity	Target Activity	Activation Condition	Target Condition
1	*Responded existence*	*Book accommodation*	*Collect tickets*	*A.Price < 30*	*T.Price > 30*
2	*Not response*	*Book transport*	*Collect tickets*	*A.Type is Bus*	*T.Price > 30*
3	*Response*	*Book transport*	*Book accommodation*	*A.Type is Bus*	*T.Price < 30*

Table 11. Example 4 constraints

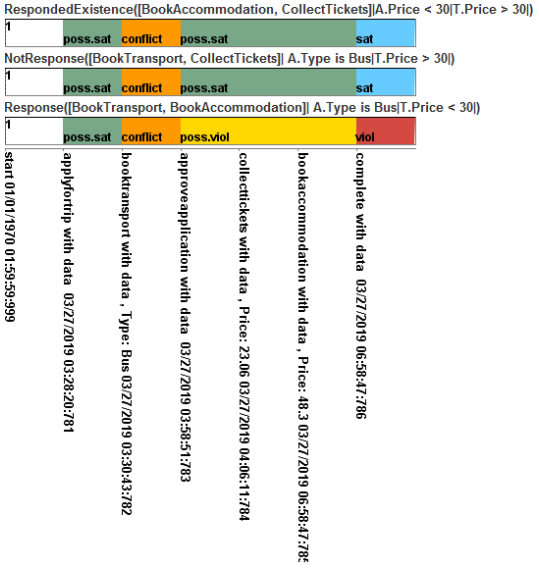


(a) Trace 1

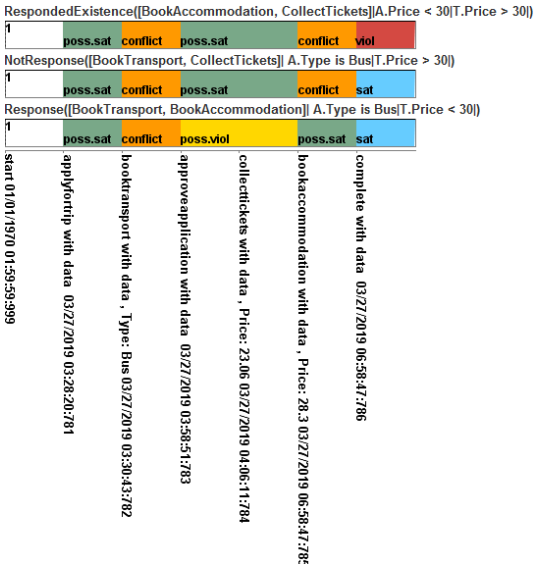


(b) Trace 2

Figure 8. Output of the monitor for Example 3



(a) Trace 1



(b) Trace 2

Figure 9. Output of the monitor for Example 4 and Traces 1 and 2

Figure 9 shows the output of the monitor for two traces. In trace 1, when *Book Transport* with *A.Type is bus* equal to *Bus* occurs then rule 2 and 3 are activated. When the constraints are checked in pairs then no conflicts will be discovered. Rules 2 and 3 are not in conflict because rule 3 can be satisfied without activity *Collect tickets* ever occurring. Rules 1 and 2 are not in conflict because rule 1 is not activated and it does not have to be activated. Constraints 1 and 3 are also not in conflict. However, since the response constraint states that activity *Book accommodation* must occur somewhere in the future with price lower than 30 and, therefore, *Collect tickets* must also occur and rule 2 says that it cannot occur means that in conjunction there will be a violation. In trace 2 it is also demonstrated that all three constraints are in conflict. In figure 10 it is shown that it is possible to have a trace where these constraints are satisfied.

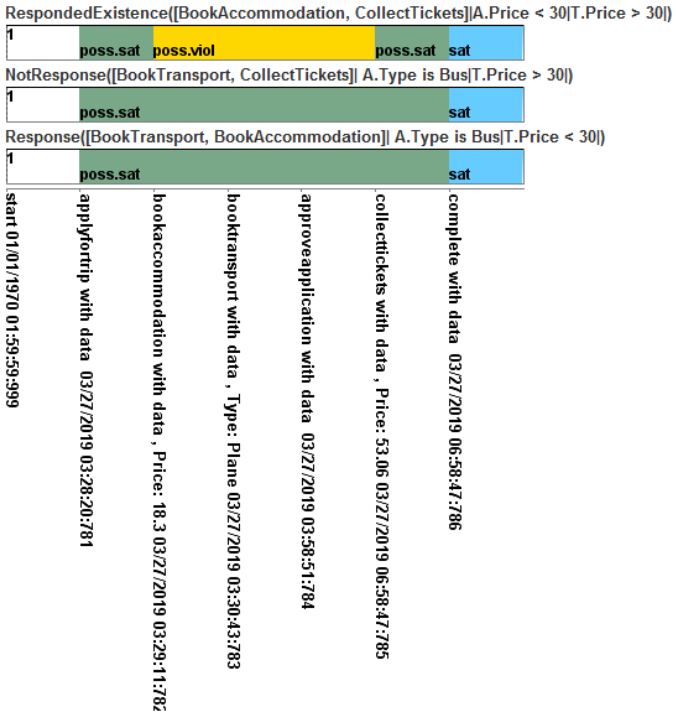


Figure 10. Output of trace 3 for example 4

7.5 Example 5

The business constraints used for this example are shown in Table 12. This example is chosen in order to demonstrate how this implementation is able to detect conflicts between constraints where constraints are indirectly related. For example, rule 1 and rule 2 have no overlapping with each other in terms of activities.

Id	Constraint	Activation Activity	Target Activity	Activation Condition	Target Condition
1	*response*	*Book transport*	*Book transport*	*A.Type is Bus*	*A.Type is not Bus*
2	*not responded existence*	*Approve application*	*Collect tickets*	-	*T.Price > 30*
3	*response*	*Book transport*	*Collect tickets*	*A.Type is Bus*	*T.Price < 30*
4	*response*	*Book transport*	*Approve application*	*A.Type is not Bus*	-

Table 12. Example 5 constraints

In the trace used for this example (Figure 11), when activity *Book transport* with *Type* equal to *Bus* occurs, a conflict is detected. Indeed, in this case, somewhere in the future there has to be an activity *Book transport* with *Type* not equal to *Bus* and an activity *Collect tickets* with *Price < 30*. Therefore, since an activity *Book transport* with *Type* not equal to *Bus* has to occur, based on rule 4, activity *Approve application* must also occur. But then, since rule 2 states that when activity *Approve application* occurs then activity *Collect tickets* with *Price > 30* cannot occur anywhere in the trace, this triggers a conflict. The conflict between these 4 constraints is detected by the tool as shown in the Fiqure 11.

7.6 Example 6

The business constraints used for this example are shown in Table 13. This constraint set was chosen because of its overall complexness.

Id	Constraint	Activation Activity	Target Activity	Activation Condition	Target Condition
1	*response*	*Book accomodation*	*Collect tickets*	*A.Price > 40*	*T.Price < 60*
2	*not responded existence*	*Book transport*	*Collect tickets*	*A.Type is Bus*	*T.Price < 60*
3	*response*	*Book transport*	*Book accomodation*	*A.Type is Plane*	*T.Price>40*
4	*init*	*Apply for trip*	-	-	-
5	*response*	*Book transport*	*Book transport*	*A.Type is Plane*	*T.Type is Bus*
6	*existence*	*Approve application*	-	-	-
7	*precedence*	*Book transport*	*Book accomodation*	-	-

Table 13. Example 6 constraints

In both traces used for this example (Figure 12) rule 3 is activated when activity *Book transport* occurs with *Type* equal to *Plane* so that *Book accomodation* with *Price > 30* should occur. In addition, rule 5 is activated also so that *Book accomodation* with

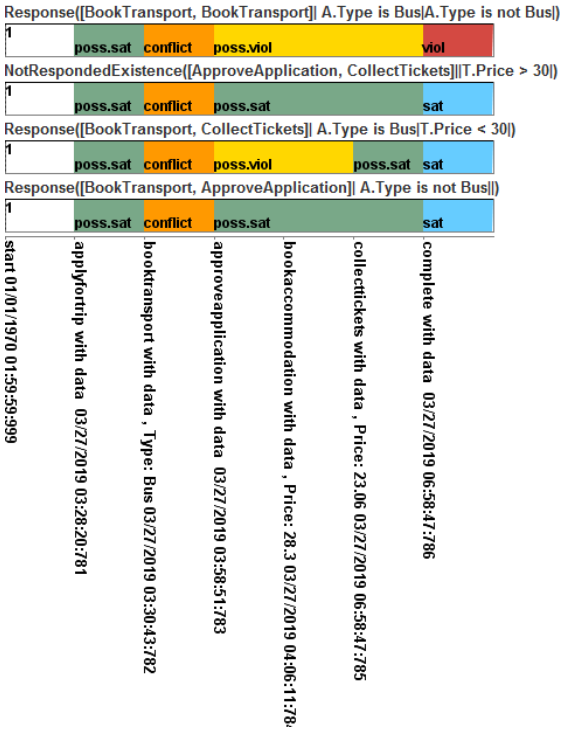
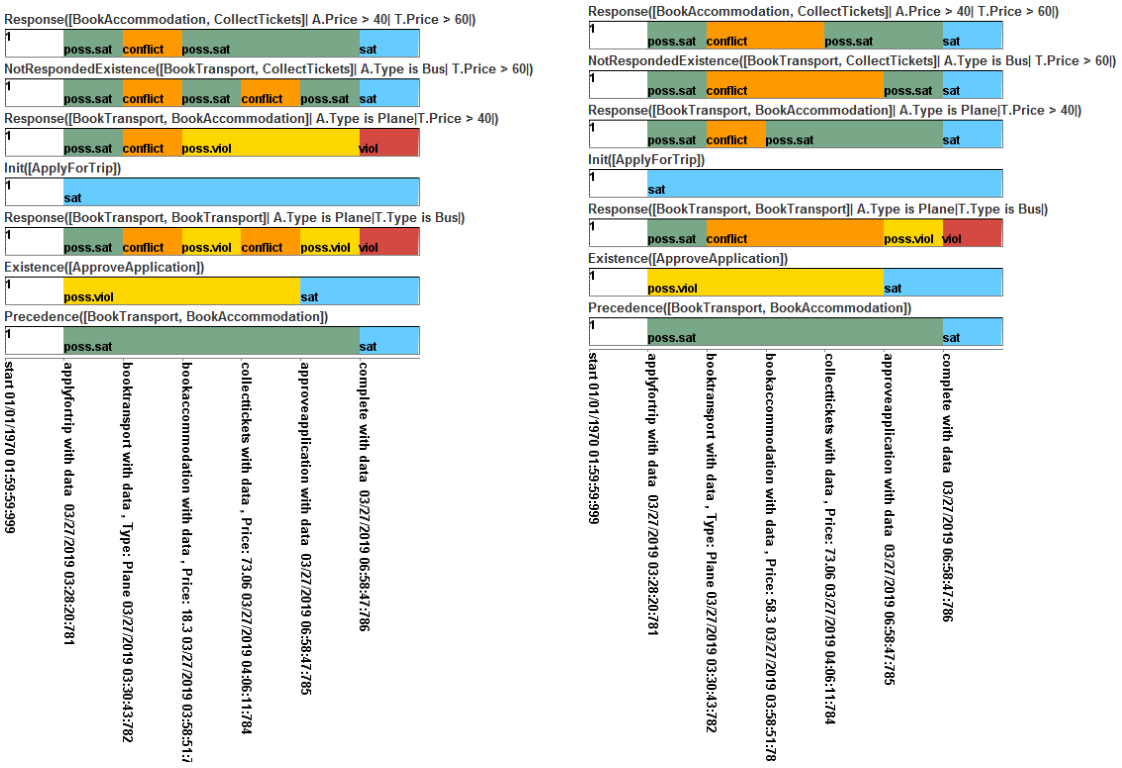


Figure 11. Output of the monitor for Example 5

Type equal to *Bus* has to occur as well. This implies that rules 1 and 2 are activated that state that *Collect tickets* with *Price* > *60* must and must not occur, respectively. This means that the four rules (1, 2 , 3 and 5) are in conflict with each other when event *Book transport* occurs, as shown in Figure 12. In addition, when activity *Collect tickets* occurs, another conflict is detected between rules 2 and 5. This is because rule 2 disallows *Book transport* with *Type* equal to *Bus*, whereas rule 5 states that it has to occur. In trace 2, the first conflict is detected in the same way as in trace 1. In addition, after that, rule 1 is activated triggering a new conflict between rules 1, 2 and 5. Also here, when activity *Collect tickets* occurs, another conflict is detected between rules 2 and 5. The monitor output for traces 1 and 2 is shown in Figure 12.

In both trace 1 and 2 rule 3 is activated when activity *Book transport* occurs where *Type is Plane* and target activity is *Book accomodation* with a condition *Price* > *30*. Rule 5 is activated where activity with the same activation and target name but the target condition is *Type is bus*. Because of rule 3 rule 1 is also involved saying that when there has to be activity name *Collect tickets* where *Price* > *60*. Because of rule 5 rule 2 must be activated as well. Rule 2 however states that there cannot be an activity *Collect tickets* where *Price* > *60* when there is *Book accomodation* where *Price* > *30*. However, Because of rules 1,3 and 5 it is stated that in order to not to violate any of the constraints both of the activities that are in rule 2 must occur. This means that the four rules are in

conflict with each other. When event *Collect tickets* occurs the Figure 12. shows conflict between rule 2 and 5. This is because Rule 2 disallows the having *Book transport* where *Type is Bus*, but rule 5 states that it has to occur. In trace 2 the rule 1 is activated and that is why it find also a conflict between rule 1,2 and 5. From Figure 12. the implementations output for trace 1 and 2 is shown.



(a) Trace 1 (b) Trace 2

Figure 12. Monitoring output of Trace 1 & 2 for example 4

7.7 Real Life example

For testing the tool on a real life event log, an event log pertaining to patient treatments in a Dutch academic hospital was chosen[3]. The Declare model used is taken from [2] and contains 16 rules. For this thesis 8 of them were selected, 4 constraints with data and 4 without dataFor this thesis 8 of them were selected, 4 constraints with data and 4 without data. Rules that were used are shown in Table 14. In Figure 13, the output obtained by replaying one of the traces of the log is shown.

[3]https://data.4tu.nl/repository/uuid:d9769f3d-0ab0-4fb8-803b-0d1120ffcf54

Id	Constraint	Activation Activity	Target Activity	Activation Condition
1	responded existence	vervolgconsult_poliklinisch	administratief_tarief_eerste_pol	-
2	response	aanname_laboratoriumonderzoek	ordertarief	-
3	responded existence	administratief_tarief_eerste_pol	aanname_laboratoriumonderzoek	-
4	not response	aanname_laboratoriumonderzoek	vervolgconsult_poliklinisch	-
5	response	administratief_tarief_eerste_pol	albumine	((A.Producer_code is SIOG) and A.Age <= 70) or ((A.Diagnosis is Maligne neoplasma cervix uteri) and (A.Diagnosis_code = 106))
6	not responded existence	telefonisch_consult	alkalische_fosfatase_kinetisch	(A.Treatment_code = 101) and ((A.Producer_code is SGAL) or (A.Producer_code is SGNA))
7	absence	aanname_laboratoriumonderzoek	-	(A.Section is Section_4) and (A.group is not General_Lab_Clinical_Chemistry) and (A.Specialism_code = 86)
8	absence	bacteriologisch_onderzoek_met_kweek_nie	-	A.group is not Medical_Microbiology

Table 14. Business rules for the hospital log

Figure 13. Output for the hospital log trace “00000578”

7.8 Comparison with the state of the art

Here, we compare our solution against the method in [25], which uses Integer Linear Programming (ILP) for conflict detection that can succesfully detect conflicts between two constraints. However, when the conflict is caused by chained triggers that involve more than two constraints the ILP method will not always succeed in detecting the conflict. In the next subsections, we present two test cases and show how our approach improves the one based on Integer Linear Programming.

7.8.1 Example 1

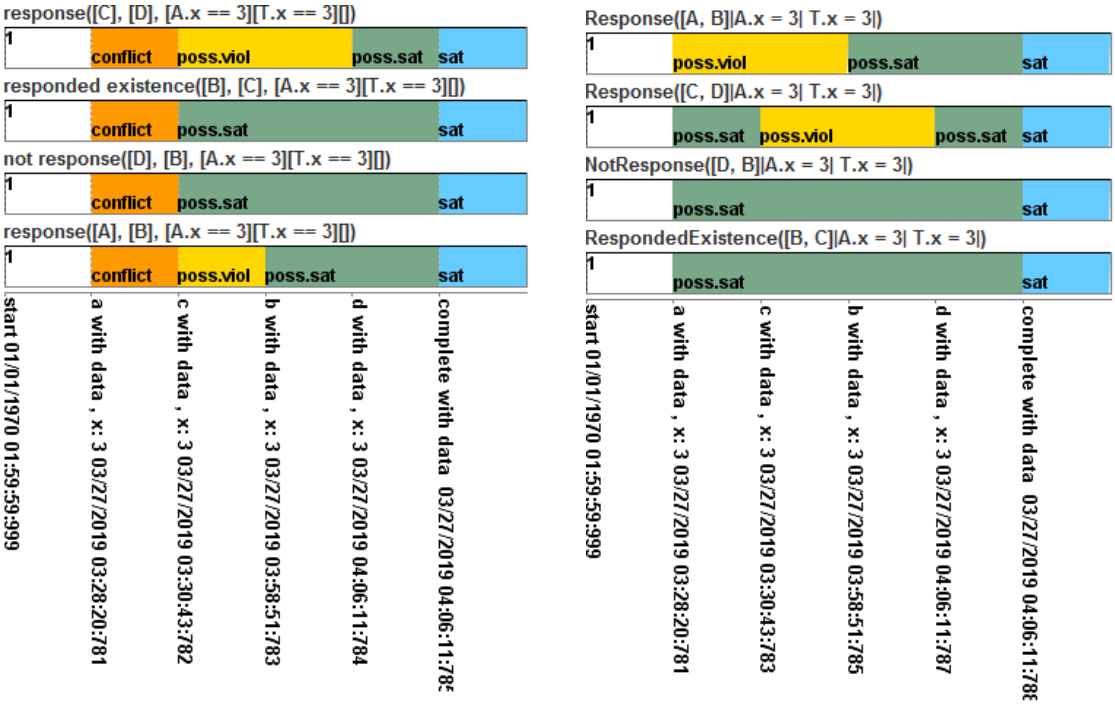
The set of constraints used for this example is shown in Table 15. When activity *A* with data *x = 3* occurs then, the ILP method will show a conflict among all the constraints as shown from Figure 14a. The occurance of *B* with *x = 3* is required because of rule 4. Therefore, *C* with *x = 3* has to occur because of rule 2 and *D* with *x = 3* because of rule 1. The execution of *D* forbids the execution of *B* with *x = 3* because of rule 3. However, this approach does not work in this case because the approach based on ILP works with obligations and permissions *without considering the temporal aspect and the order of execution of the activities*. For example, in the above case, there is a sequence where all the constraints are satisfied which is *A, C, B, D*. The existence of this combination is detected from Alloy. Indeed, as shown in Figure 14b the Alloy method does not trigger the conflict since it is able to find a sequence that satisfies all the constraints.

Id	Constraint	Activation Activity	Target Activity	Activation Condition	Target Condition
1	*response*	*C*	*D*	*A.x = 3*	*T.x = 3*
2	*responded existence*	*B*	*C*	*A.x = 3*	*T.x = 3*
3	*not response*	*D*	*B*	*A.x = 3*	*T.x = 3*
4	*response*	*A*	*B*	*A.x = 3*	*T.x = 3*

Table 15. Constraints of the first comparision example

7.8.2 Example 2

The set of constraints used for this example is shown in Table 16. When activity *A* occurs with *Type = Car* and then activity *C* with *Type = Bus* then activity *B* has to occur two times, once with *Type = Bus* and once with *Type = Car*. This already triggers a conflict. Indeed, when activity *B* is executed with *Type = Car* (or *Bus*), activity *D* with *Type = Car* (or *Bus* respectively) has to be executed immediately after. This event triggers rule 3 which disallows the occurrence of *B* with *Type = Bus* (or *Car* respectively). Therefore, when the activity *C* occurs, there is no trace that would satisfy all the constraints. Such complex interplay among constraints cannot be detected by the ILP method that is for this



(a) ILP solution

(b) Alloy solution

Figure 14. Monitoring output of Alloy and ILP solutions for example 1

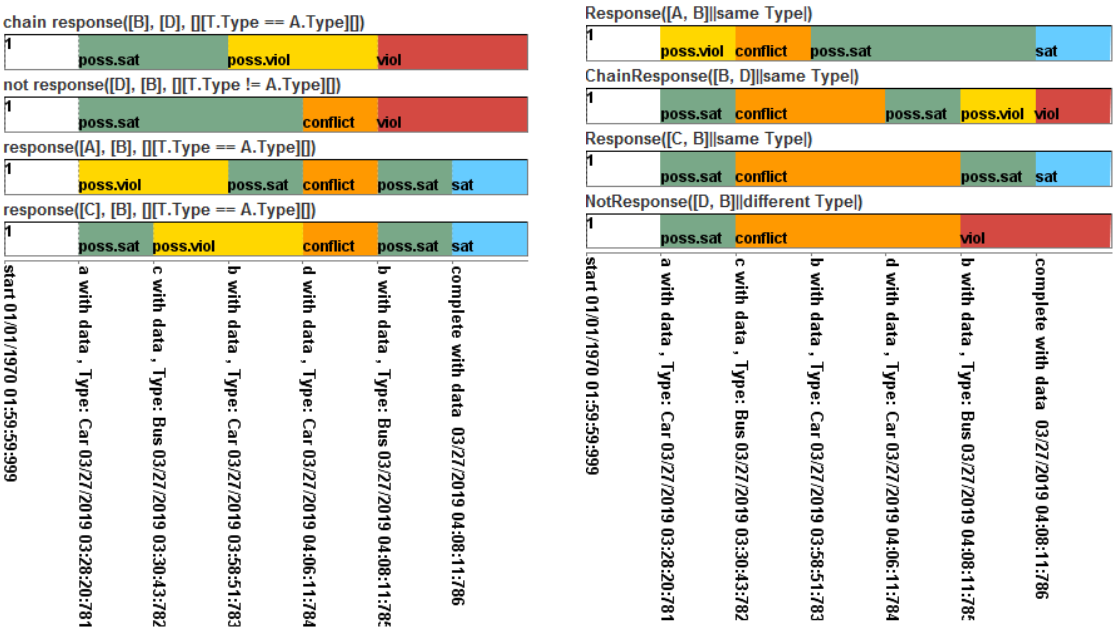
reason proven to be able to detect only conflicts due to the interaction of two constraints. Indeed, as shown in Figure 15a the ILP method is unable to detect the conflict among the constraints when activity *C* occurs. On the other hand, the Alloy solution is able to detect the conflict as shown in Figure 15b

Id	Constraint	Activation Activity	Target Activity	Activation Condition	Target Condition
1	*chain response*	*B*	*D*	-	*Same type*
2	*response*	*A*	*B*	-	*Same type*
3	*response*	*C*	*B*	-	*Same type*
4	*not response*	*D*	*B*	-	*Same type*

Table 16. Constraints of the second comparision example

7.9 Performance

Figure 16 shows the processing time for each event for one of the traces in the hospital log considered before. The blue line is related to a model with 8 constraints, the gray line is related to a model containing 4 constraints with data, and the orange line is related



(a) ILP solution

(b) Alloy solution

Figure 15. Monitoring output of Alloy and ILP solutions for example 2

to a model containing 4 constraints without data. An equal number of constraints with and without data were considered in order to see whether there is a significant difference between processing times. The chart shows that monitoring constraints without data takes less time. Also the size of the constraint set matters, as monitoring a set of 8 constraints is more time-consuming than monitoring a constraint set with 4 constraints. This difference is more evident when the partial trace contains more events. For a partial trace with 40 events, processing an event using a constraint set of size 8 takes more than 6 seconds.

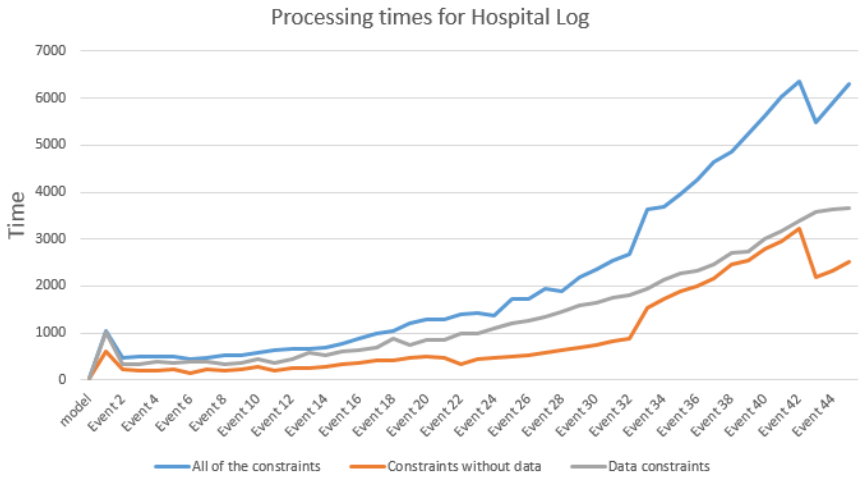


Figure 16. Processing times in milliseconds for every event

8 Conclusion and future work

In Section 4 we stated that this thesis addresses the following research questions:

1. How to use Alloy to monitor individual data-aware constraints?
2. How to use Alloy to detect conflicts due to the interplay of data-aware constraints?
3. Is this approach applicable to real-life case studies?

We have shown how to encode a MP-Declare model and a stream of events in Alloy in order to monitor individual constraints at runtime. We have also presented an approach based on Alloy to detect conflicts due to the interplay of two or more constraints. This approach was also successfully validated by using a real life event log showing that the approach is applicable to monitor traces that are sufficiently long. The approach was also compared to an existing solution based on ILP that was not able to detect conflicts caused by chained triggers due to the interaction of more than two constraints. In particular, we have shown that the Alloy-based solution was able to detect these complex conflicts that the ILP solution was either not able to detect as early as possible or not able to detect at all.

There are some limitations of the proposed approach that we would like to investigate as future work. First, the processing time during the monitoring may go high when the partial trace is very long and the set of constraints is large. Even if this is not a problem when monitoring business processes where the response time is not crucial, this could become an issue when using the monitor in real-time systems. This problem could be solved to keep the "state" of a certain constraint with respect to a trace in memory without encoding the entire partial trace in Alloy every time a new event becomes available.

Another issue is that, to detect conflicts, we need to specify an upper bound indicating the maximum length of the possible continuations of the partial trace under analysis. This is needed to ensure that if a certain set of constraint is not satisfiable for any continuation of the partial trace within that bound, then the set of constraint is not satisfiable for any finite continuation of the partial trace. Even if this is not an issue if some background knowledge can be used (for example in the form of a process model) to determine the bound, if the bound is not known apriori, it has to be computed for every combination of constraints to be monitored. In this thesis, we computed this bound when monitoring individual constraints. The proof that this bound exists and can be derived for any combination of Declare constraints was not in the scope of this thesis.

The tool provided in this thesis does not deal with time conditions available in MP-Declare. This is because monitoring time conditions with Alloy is a very difficult task since Alloy does not natively support numbers. Monitoring time conditions could be implemented with an ad hoc implementation that could complement the Alloy-based solution. However, even if this type of implementation is easy to develop when

monitoring individual constraints, the detection of conflicting constraints is an inherently difficult problem.

References

[1] Norris Syed Abdullah, Shazia Sadiq, and Marta Indulska. Emerging challenges in information systems research for regulatory compliance management. In Barbara Pernici, editor, *Advanced Information Systems Engineering*, pages 251–265, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[2] Linh Thao Ly, Fabrizio Maria Maggi, Marco Montali, Stefanie Rinderle-Ma, and Wil M.P. van der Aalst. Compliance monitoring in business processes: Functionalities, application, and tool-support. *Information Systems*, 54:209 – 234, 2015.

[3] Andrea Burattin, Fabrizio M. Maggi, and Alessandro Sperduti. Conformance checking based on multi-perspective declarative process models. *Expert Systems with Applications*, 65:194 – 211, 2016.

[4] Denizalp Kapisiz. Rule mining with rum. Master's thesis, UNIVERSITY OF TARTU, 2019.

[5] Wil M. P. Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*, volume 136. 01 2011.

[6] H. M. W. Verbeek, Joos C. A. M. Buijs, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Xes, xesame, and prom 6. In Pnina Soffer and Erik Proper, editors, *Information Systems Evolution*, pages 60–75, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[7] Vasyl Skydanienko. Data-aware synthetic log generation for declarative process models. Master's thesis, University of Tartu, 2018.

[8] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Oct 1977.

[9] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[10] OMG. Business Process Model and Notation (BPMN), Version 2.0, January 2011.

[11] W. M. P. van der Aalst, M. Pesic, and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - Research and Development*, 23(2):99–113, May 2009.

[12] A. Burattin, F. M. Maggi, W. M. P. van der Aalst, and A. Sperduti. Techniques for a posteriori analysis of declarative processes. In *2012 IEEE 16th International Enterprise Distributed Object Computing Conference*, pages 41–50, Sep. 2012.

[13] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[14] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[15] Jonathan E. Cook and Alexander L. Wolf. Software process validation: Quantitatively measuring the correspondence of a process to a model. *ACM Trans. Softw. Eng. Methodol.*, 8(2):147–176, April 1999.

[16] J. E. Cook, Cha He, and Changjun Ma. Measuring behavioral correspondence to a timed concurrent model. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 332–341, Nov 2001.

[17] A. Rozinat and W.M.P. van der Aalst. Conformance checking of processes based on monitoring real behavior. *Information Systems*, 33(1):64 – 95, 2008.

[18] A. Adriansyah, B. F. van Dongen, and W. M. P. van der Aalst. Conformance checking using cost-based fitness analysis. In *2011 IEEE 15th International Enterprise Distributed Object Computing Conference*, pages 55–64, Aug 2011.

[19] Massimiliano de Leoni, Fabrizio M. Maggi, and Wil M.P. van der Aalst. An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data. *Information Systems*, 47:258 – 277, 2015.

[20] Massimiliano de Leoni, Jorge Munoz-Gama, Josep Carmona, and Wil M. P. van der Aalst. Decomposing alignment-based conformance checking of data-aware process models. In Robert Meersman, Hervé Panetto, Tharam Dillon, Michele Missikoff, Lin Liu, Oscar Pastor, Alfredo Cuzzocrea, and Timos Sellis, editors, *On the Move to Meaningful Internet Systems: OTM 2014 Conferences*, pages 3–20, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[21] Elham Ramezani Taghiabadi, Vladimir Gromov, Dirk Fahland, and WilM P van der Aalst. Compliance checking of data-aware and resource-aware compliance requirements. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 237–257. Springer, 2014.

[22] Diana Borrego and Irene Barba. Conformance checking and diagnosis for declarative business process models in data-aware scenarios. *Expert Systems with Applications*, 41(11):5340–5352, 2014.

[23] Fabrizio Maria Maggi, Michael Westergaard, Marco Montali, and Wil M. P. van der Aalst. Runtime verification of ltl-based declarative process models. In Sarfraz

Khurshid and Koushik Sen, editors, *Runtime Verification*, pages 131–146, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[24] Marco Montali, Fabrizio Maria Maggi, Federico Chesani, Paola Mello, and Wil M. P. van der Aalst. Monitoring business constraints with the event calculus. *ACM TIST*, 5:17:1–17:30, 2013.

[25] Ubaier Ahmad Bhat. Runtime monitoring of data-aware business rules with integer linear programming. Master's thesis, UNIVERSITY OF TARTU, 2016.

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Karl-Kristjan Kokk**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Compliance Monitoring of Data-Aware Declarative Process Models

   supervised by Fabrizio Maria Maggi

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Karl-Kristjan Kokk
15.05.2020