UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Timothy Iyanuoluwa Fadayini

A Visual Editor for the Declare Process Modelling Language

Master's Thesis (30 ECTS)

Supervisor(s): Anti Alman, MSc
Fabrizio Maggi, PhD

Tartu 2023

A visual editor for the Declare process modelling language

Abstract:
Business process modelling represents information flow, business activities, and decision logic in business processes. Process models can be divided into at least two types, procedural and declarative process models. Procedural models aim to describe end-to-end processes and only allow for activities explicitly triggered through control flow. However, this can quickly become cumbersome if the process is loosely-structured or includes many varieties. For example, a hospital process can quickly become unreadable because of the various paths representing the large variety of possible clinical pathways to be specified in the model. Cases like this are where the declarative modelling approach could be a better choice. Declarative modelling allows modellers to capture constraints on the allowed activity flows, so if flows do not violate the specified constraints, they are allowed. There are multiple applications available for working with process models. For example, Disco and Apromore are for procedural models, and RuM is for declarative models. This thesis work focuses on RuM. More specifically, it builds on the Master's Thesis of A. Alman, 'A Desktop Application for Advanced Business Rule Mining', to further improve RuM by developing a new visual process model editor for the Declare process modelling language. This new editor is developed from the ground up, with the aim of replacing the current, mainly table-based, editor with a more appealing, easy-to-use, and fluid model editor where the user can work with the graphical representation of the Declare model directly, thus resolving one of the more difficult future works outlined in the thesis of A. Alman. A user evaluation with experts was conducted to evaluate the final resulting editor, and the main findings of this evaluation are presented as a part of the thesis.

Keywords:
Process mining, Declare models, Constraints, Graphical editor, Layouting.

CERCS: P170 - Computer science, numerical analysis, systems, control

Visuaalne redigeeria äriprotsesside modelleerimise keelele Declare

Lühikokkuvõte: Äriprotsesside modelleerimine esindab infovoogu, äritegevust ja otsustusloogikat äriprotsessides. Protsessimudelid võib jagada vähemalt kaheks tüübiks, protseduurilisteks ja deklaratiivseteks protsessimudeliteks. Protseduuriliste mudelite eesmärk on kirjeldada protsesse otsast-lõpuni ning sellest tulenevalt on nende mudelite põhjal lubatud ainult need tegevused, mis on selgesõnaliselt käivitatud juhtimisvoo kaudu. See võib aga kiiresti muutuda tülikaks, seda eriti juhul kui protsess on lõdvalt struktureeritud või hõlmab paljusid variante. Näiteks võib haiglaprotsess kiiresti muutuda loetamatuks, kuna mudelis patsiendi ravi täielik modelleerimine eeldaks kõikvõimalike kliiniliste teide modelleerimist. Sellistel juhtudel on deklaratiivne modelleerimine sageli parem valik. Deklaratiivne modelleerimine võimaldab modelleerijatel töövoogude täieliku otsast-lõpuni kirjeldamise asemel keskenduda lubatud tegevusvoogude kitsendustele, ning lähtub põhimõttest et kui mingi töövoog ei riku neid kitsendusi siis see töövoog on lubatud. Protsessimudelitega töötamiseks on saadaval mitmeid rakendusi, näiteks Disco ja Apromore protseduuriliste mudelite, ning RuM deklaratiivsete mudelite jaoks. See lõputöö keskendub RuM-ile. Täpsemalt tugineb see A. Almani magistritööle "Töölauarakendus täiustatud ärireeglite kaevandamiseks", et täiustada RuM-i veelgi, töötades välja uue visuaalse protsessimudeli redaktori protsesside modelleerimiskeele Declare jaoks. See uus redaktor on välja töötatud algusest peale eesmärgiga asendada praegune, peamiselt tabelipõhine redaktor, atraktiivsema, lihtsamini kasutatava ja sujuvama mudeliredaktoriga, kus kasutaja saab töötada otse Declare mudeli graafilise esitusega. Selle kaudu lahendab käesolev lõputöö ühe raskema A.Almani lõputöös välja toodud võimalikest RuM rakenduse edasiarendustest. Käesoleva lõputöö käigus valminud mudeli redigeerija hindamiseks viidi kasutustestid kümne valdkonna eksperdiga ning nende testide peamised järeldused on esitatud käesoleva lõputöö osana.

Võtmesõnad:
Protsessikaeve, deklaratiivsed mudelid, kitsendused, graafiline redaktor, paigutus.

CERCS: P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

Process mining is a data-driven technique which involves extracting knowledge and information from event log data with the aim of discovering, monitoring and improving business processes. Process mining is usually divided into three main branches, which are process discovery, conformance checking, and process enhancement [1]. Process discovery refers to automatically generating a process model based on an event log (i.e., a recording of the performed activities) of the process. Conformance checking is used to find any discrepancies between an event log and a process model. Process enhancement refers to modifications of a process model based on the information gained from an event log with the goal of improving the process performance w.r.t. some performance metrics. These three main branches of process mining either create, interact with, or modify process models.

Process models can be divided into at least two types, procedural and declarative process models, with procedural models being the most common type. Procedural models aim to describe end-to-end processes and only allow for activities that are explicitly triggered through control flow (often tracked via some form of token semantics) [29]. However, this approach can become cumbersome very quickly if the process is loosely-structured or includes a high number of variations. For example, a model of a hospital process can quickly become unreadable because of the various paths that should be represented in the model to account for all possible treatment options. The declarative modelling approach could be a better choice for cases like these. Declarative modelling allows modellers to capture constraints of the allowed activity flows, meaning that if flows do not violate the specified constraints, they are allowed [13]. This leads to more flexibility in the process descriptions, as every possible option does not have to be explicitly represented in the model.

There are multiple process mining applications available today. For example, Disco[1] and Apromore[2] for procedural models and RuM[3] [23] for declarative models. One of the prominent declarative process modelling languages is Declare [22]. The focus of this thesis is on RuM, a rule-mining application for declarative process mining. RuM is a desktop application that provides a comprehensive declarative process mining toolset that is easy to use for both beginners and experts in process mining. RuM consolidates multiple Declare-based process mining methods, which have been packaged into a single application that supports the interoperability of all these methods. RuM currently has the following five major features:

- Process discovery – RuM has the ability to generate a model from the data found

[1]https://fluxicon.com/disco/
[2]https://apromore.com/
[3]https://rulemining.org/

in the event log while making minimal assumptions about the specific data being provided.

- Conformance checking – RuM can also check the modelled behaviour and compare this to the observed behaviour of the process.
- MP-Declare Editor – RuM includes a model editor to define Declare constraints, based on which a non-interactive visual representation of the model is automatically generated.
- Log generation – RuM also has the ability to generate artificial logs based on pre-existing models.
- Monitoring – RuM monitors animate a Declare model by replaying a log over it and showing how each individual event affects the satisfaction or violation of each individual constraint.

This thesis builds on the Master's Thesis of A. Alman [2] to significantly improve the process modelling capabilities in RuM. More specifically, the goal of this thesis is to develop an appealing, easy-to-use visual editor from the ground up for the Declare process modelling language. Although in the thesis work of A. Alman, the application was redesigned to be more user-friendly and visually appealing than the initial version of RuM by D. Kapisiz [17], the editor nevertheless remained table-based and heavily reliant on a library called VisJs[4]. This means that modelling was done by adding values to rows in a table, and then a non-interactive visualisation of the model was generated from those values. Compared to, for example, the BPMN editor in Apromore, which is much more fluid and interactive, the editor in RuM simply refreshed a static visualisation of the model after each modification of the model. This was not in line with the expectations of most users, thus necessitating a redesign of the editor in RuM to better match the user's current expectation and further introduce ease of use, fluidity and overall better user experience when designing models. This resulted in the first visual editor for the MP-Declare notation, which is also the main contribution of this thesis. Additionally, a new code architecture has been developed for the editor in order to make it easier to further improve in the future.

A user evaluation with experts was conducted to evaluate the final resulting editor, and this evaluation's main findings are presented as part of the thesis. As part of this evaluation, the participants were provided with specific modelling tasks and the newly developed RuM editor. The tasks were performed on the newly developed editor with minimal interventions by the user evaluation team, and an anonymous survey was completed by the participants afterwards.

[4]https://github.com/mdaines/viz-js

The rest of this thesis is organised as follows.[5] In the second chapter, we give an overview of the process mining field and highlight some basics of the Declare modelling language. In the third chapter, we describe RuM in detail, shedding light on the features of RuM, the previous Declare editor, and the application architecture. The third chapter finishes with an overview of related work. The fourth chapter highlights the method used to achieve the objective of developing a new Declare editor. In the fifth chapter, we outline the requirements, discuss the technology stack, and provide a detailed description of the core components of the new editor. We then discuss the results and findings of the user evaluation of the new editor in chapter six and conclude the thesis in chapter seven.

[5]While writing this thesis, Grammarly (available at: `https://www.grammarly.com/`, an AI writing assistant, was used to verify the written work's spelling, punctuation and grammar.

2 Background

In this chapter, we will give an overview of what process mining is and the various branches of process mining. The chapter concludes with an overview of the Declare modelling language and its visual representation.

2.1 Process Mining

Process mining is a sub-field of Business Process Management (BPM) that aims to discover, monitor and improve business processes based on how they are executed in reality (i.e., not assumed processes) by extracting information from event logs readily available in today's systems [28]. Process mining addresses the challenge of having a truly "closed" BPM life cycle by establishing links between the process models, the actual processes and their data. Knowledge necessary for process mining can be extracted from a variety of systems today, such as Customer Relationship Management (CRM) systems, Enterprise Resource Planning (ERP) systems, and Product Data Management (PDM) systems.

There are three main branches of process mining, namely process discovery, conformance checking and process enhancement. To apply any of these three branches, an event log with information on the actual process executions has to be present. An event log is assumed to consist of time-stamped events (activities performed) in the order in which they were performed, with each event belonging to a single instance of process execution. Event logs can be represented in various formats, but the most used format by process mining tools is the eXtensible Event Stream (XES)[6] format. This is because XES is a standardized, and therefore less error-prone, versatile file format for storing and exchanging event logs [14].

All the three above-mentioned branches of process mining involve either creating or modifying a process model, which is where the Declare editor comes into play. For example, automatically discovered models may need some modifications due to flaws resulting from data quality issues, for conformance checking, it may be needed to modify the model to focus on checking only specific parts (sets of constraints in the case of Declare) of the process model, and for process improvement, a to-be process model is often created before any process changes are implemented in practice.

2.1.1 Process Discovery

This technique takes an event log and generates a model without any prior information about the specific process and minimal assumptions on what data about the process is provided. For example, if the event log contains resource information, then resource-related models can also be discovered, however, that information is not required. Process

[6]http://www.xes-standard.org/

discovery is the most prominent and challenging process mining technique [29]. Because models can be generated without prior information, process discovery algorithms are domain-independent, meaning that the same process discovery algorithm can be reused in any domain to discover process models.

In addition to control-flow discovery (modelling sequence of individual activities), process mining can also be used to discover business rules, policies and organizational models. The discovery of declarative models is the focus of RuM. Declarative models do not attempt to capture or model the process from end to end. Instead, they model the concrete rules (constraints) the process adheres to using a process modelling language called Declare. The model generated from process discovery can be viewed and, if needed, also modified using the Declare editor.

2.1.2 Conformance Checking

Conformance checking is used to compare a model behaviour to the observed behaviour of the process as recorded in the event log. Conformance checking is used to check if reality, as recorded in the event log, matches the model and vice versa.

The goal is to find commonalities and discrepancies between the modelled behaviour and the observed behaviour. Conformance checking is relevant for business alignment and auditing. Information about these deviations can be used as a starting point to identify deviating traces, find inefficiencies, or judge the quality of a discovered model.

2.1.3 Process Enhancement

Process enhancement aims to enhance a process model based on information retrieved from event logs. Usually, this involves “extending”, improving, or otherwise modifying an existing process model based on the information gained from analysing previous process executions.

One type of enhancement is repair, i.e., modifying a model to reflect reality better. For example, if two activities in a model are modelled sequentially but, in reality, can happen in any order, then the model can be corrected to reflect this. Another type of enhancement is “extension”, which means adding a new perspective to the process model by correlating it with the log. An example is the extension of a process model with additional performance data.

2.2 Basics of Declare Modelling Language

Declare is a declarative process modelling language wherein a process is specified via a set of constraints between activities such that each constraint must be satisfied by every execution of the process [13]. This enables a constraint-based approach to define a loosely-structured process model [22]. Declare is grounded in temporal logic (more specifically, Linear Temporal Logic for finite traces, LTL_f) [15], which allows for the development of formally sound and executable process models.

The aim of Declare is to capture important aspects of the process [2]. Consider, for example, a situation where if activity “A” is executed for a particular process instance, then activity “B” cannot be executed in the same instance and vice versa. In procedural process models, every possible execution matching this rule would need to be modelled explicitly, resulting in an over-specified process model, which, depending on the overall structure of the process, may easily become unreadable. But by using Declare, this rule can be modelled by simply specifying the connection between “A” and “B” with a single constraint that matches the exclusion condition.

The main building blocks of the Declare language are constraints. Constraints consist of a template (the constraint type) and a reference to one or two activities. A template defines the semantic meaning of the constraint, while the activity references define to which activities this meaning applies.

There are three main concepts to understand when working with Declare constraints: constraint activation, constraint fulfilment, and constraint violation. Constraint is considered activated if the activation activity occurs in the process execution. An activation triggers some obligation on the occurrence of another activity (the target) in the same process instance. By the end of the process, all the activated constraints must be fulfilled. If that is not the case (i.e., at least one constraint is violated), then the execution is considered to not match the given model. The Declare paradigm was extended to include data conditions and called MP-Declare(Multi-Perspective Declare). The data conditions allow modellers to specify conditions that govern the constraint. There are three types of data conditions: activation condition, correlation condition and time condition. These data condition types will be explained further in Section 3.2.4.

Each template of the Declare language belongs to one of the following groups [2]:

- Unary templates – They refer to a single activity. They are used to define the cardinality of an activity in a process or to define where in the process the activity should occur. Examples of these templates are shown in Table 1.

- Positive binary templates – They refer to two activities and are used to define a “positive” relation between two activities. A positive relation means that when an activation occurs, the target will also occur in some specific relation to the activation. Examples of these templates are be found in Table 2.

- Negative binary templates – They refer to two activities and are used to define a “negative” relation between two activities. A negative relation means that when an activation occurs, then the occurrence of the target is restricted in some way. Examples of these templates are shown in Table 3.

- Choice templates – They refer to two activities. They are additional binary constraints which do not fit into the positive and negative binary template. There are two of them, Exclusive Choice and Choice.

Table 1. Semantics of some unary Declare templates.

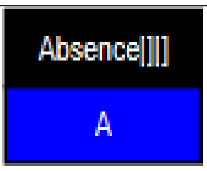
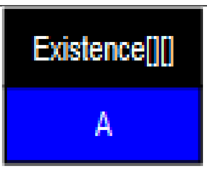
Template	**Explanation**	**Notation**
Absence[A]	Activity A does not occur	
Existence[A]	Activity A occurs at least once	

Table 2. Semantics of some positive Declare templates.

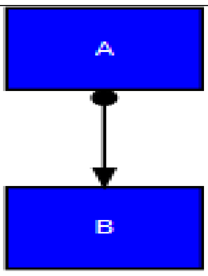
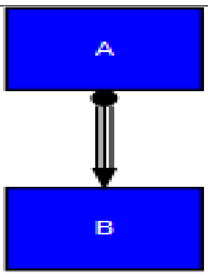
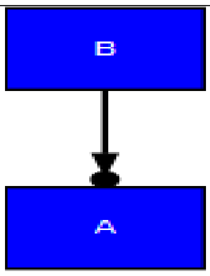
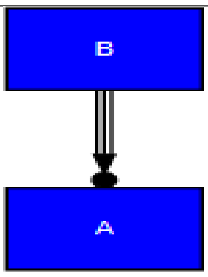
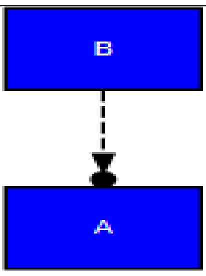
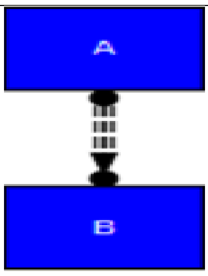
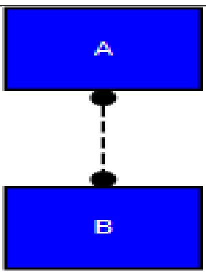
Response[A, B]	If Activity A occurs, then Activity B occurs after Activity A	
Chain Response[A, B]	Each time Activity A occurs, then Activity B occurs immediately afterwards	
Precedence[A, B]	Activity A occurs if preceded by Activity B	
Chain Precedence[A, B]	Each time Activity A occurs, then Activity B occurs immediately beforehand	

Table 3. Semantics of some negative Declare templates.

Not Precedence[A, B]	Activity A occurs if it is not preceded by activity B	
Not Chain Succession[A, B]	Activity A and Activity B occur together if and only if the latter does not immediately follow the former	
Not Co-Existence[A, B]	Activity A and activity B never occur together	

3 RuM, the rule mining application

In this chapter, we introduce the RuM application. We start by highlighting its declarative process mining features, which are then followed by a more detailed overview of the Declare editor that is currently available in the public version of RuM. We continue with a description of the architecture, the software design, and the technology used thus far for developing the application RuM. The chapter concludes with a description of how the application is packaged and an overview of related work.

3.1 Features of RuM

At the time of writing, the publicly available version of RuM (version 0.6.9) has five major features: process discovery, conformance checking, Declare editor, log generation and monitoring. These features, as well as their sub-components and interactions between them, are shown in Figure 1. In the following, we give a brief overview of all the features currently available in RuM, except the editor, which will be discussed in more detail in Section 3.2.

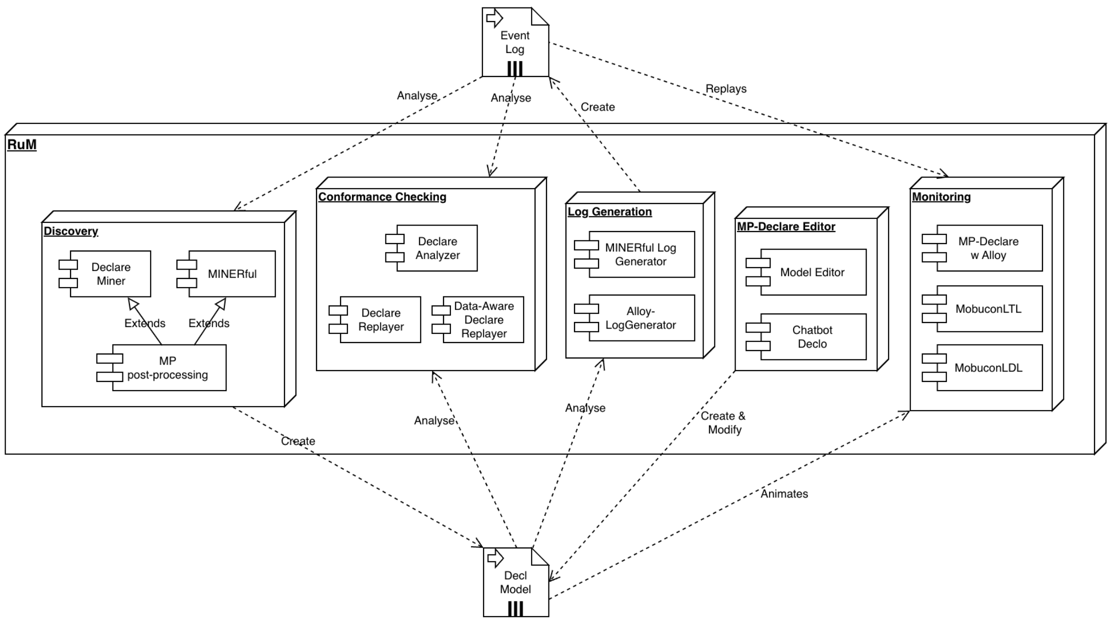
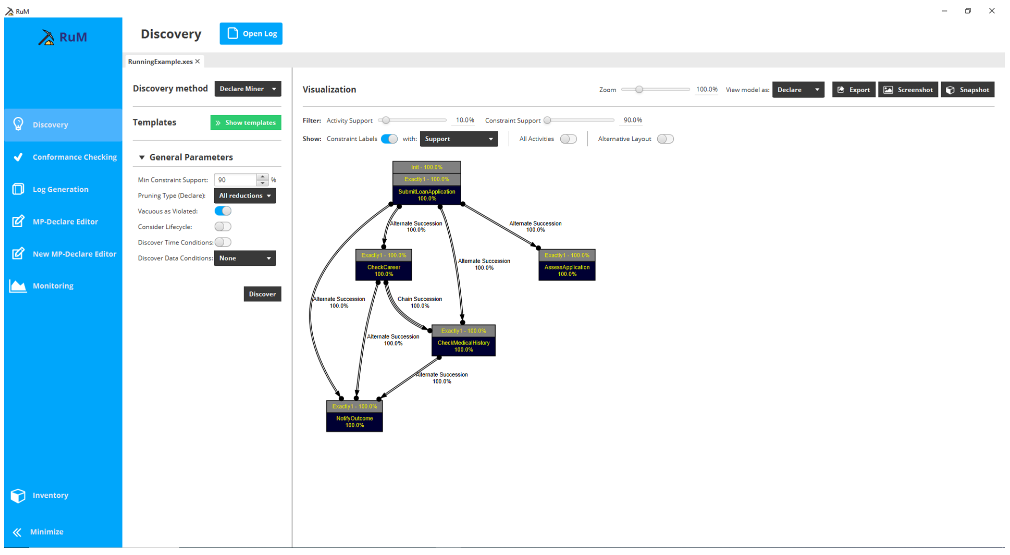


Figure 1. Main features in RuM.

3.1.1 Process discovery

This refers to generating a process model using only the data found in an event log. Process discovery makes minimal assumptions about the input data and the underlying process and aims at capturing the behaviour and patterns seen in the event log and faithfully representing them in a resulting process model. Four methods are available for process discovery: Declare Miner, MINERful, MP-Declare Miner and MP-MINERful [6]. The methods prefixed with 'MP' use the same process mining algorithms as implied in their names but with an additional post-processing step to enrich the discovered model with data conditions also based on the same event log. The result of process discovery can be presented in three different views in RuM: Declare view, Textual view, and Automaton view, with the default view being the Declare view. Figure 2 gives an example of the Declare view. All views show the same process model but emphasise different details and aspects. The Declare view, as the name implies, shows the Declare process model, the Textual view shows a textual description of the semantics of each constraint in the Declare model, and the Automaton view shows the automaton translation of the LTL_f semantics of Declare model. Filtering of activity and constraints are supported on all views.



Discovery method: Declare Miner

Min Constraint Support: 90 %
Pruning Type (Declare): All reductions
Vacuous as Violated: on
Consider Lifecycle: off
Discover Time Conditions: off
Discover Data Conditions: None

Zoom 100.0% View model as: Declare

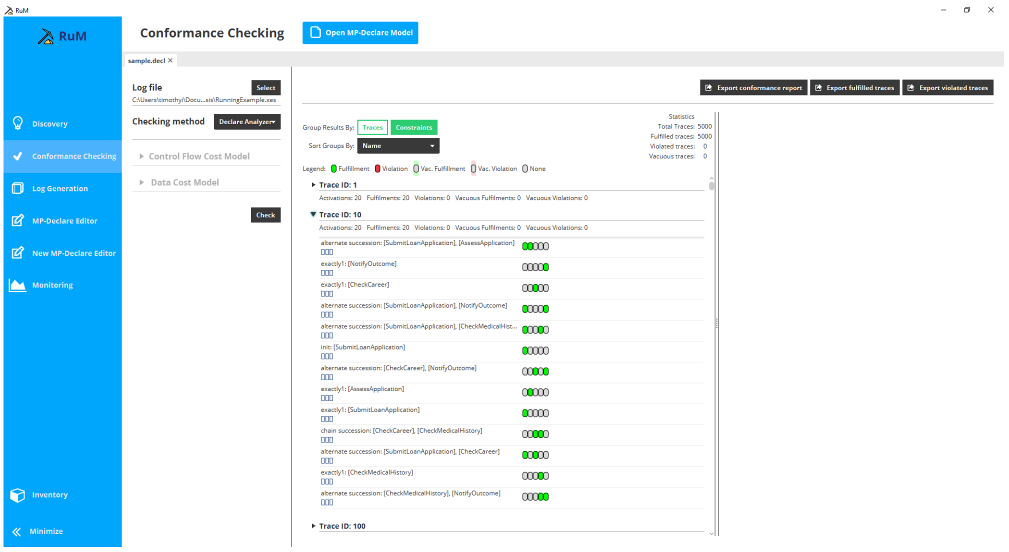
Filter: Activity Support 10.0% Constraint Support 90.0%

Show: Constraint Labels on with: Support | All Activities off | Alternative Layout off

Figure 2. Process discovery view.

3.1.2 Conformance Checking

Conformance checking supports three methods: Declare Analyzer, Declare Replayer and Data-Aware Declare Replayer [6]. The conformance-checking results are presented



Conformance Checking

sample.decl

Log file: C:\Users\timothyi\Docu...sis\RunningExample.xes

Checking method: Declare Analyzer

Group Results By: Constraints

Sort Groups By: Name

Statistics
Total Traces: 5000
Fulfilled traces: 5000
Violated traces: 0
Vacuous traces: 0

Trace ID: 1
Activations: 20 Fulfilments: 20 Violations: 0 Vacuous Fulfilments: 0 Vacuous Violations: 0

Trace ID: 10
Activations: 20 Fulfilments: 20 Violations: 0 Vacuous Fulfilments: 0 Vacuous Violations: 0

alternate succession: [SubmitLoanApplication], [AssessApplication]
exactly1: [NotifyOutcome]
exactly1: [CheckCareer]
alternate succession: [SubmitLoanApplication], [NotifyOutcome]
alternate succession: [SubmitLoanApplication], [CheckMedicalHist...
init: [SubmitLoanApplication]
alternate succession: [CheckCareer], [NotifyOutcome]
exactly1: [AssessApplication]
exactly1: [SubmitLoanApplication]
chain succession: [CheckCareer], [CheckMedicalHistory]
alternate succession: [SubmitLoanApplication], [CheckCareer]
exactly1: [CheckMedicalHistory]
alternate succession: [CheckMedicalHistory], [NotifyOutcome]

Trace ID: 100

Figure 3. Conformance checking view.

in groups, with each group representing the results for a specific trace[7] or constraint. The Declare Analyzer takes a model and an event log as inputs and returns activations, violations, and fulfillments in the log for each constraint in the model. While the Declare Replayer and the Data-Aware Declare Replayer report trace alignments (i.e., traces that are modified to match the behaviour specified in the model). The conformance checking results are presented in one view divided into two panels as shown in Figure 3.

3.1.3 Log generation

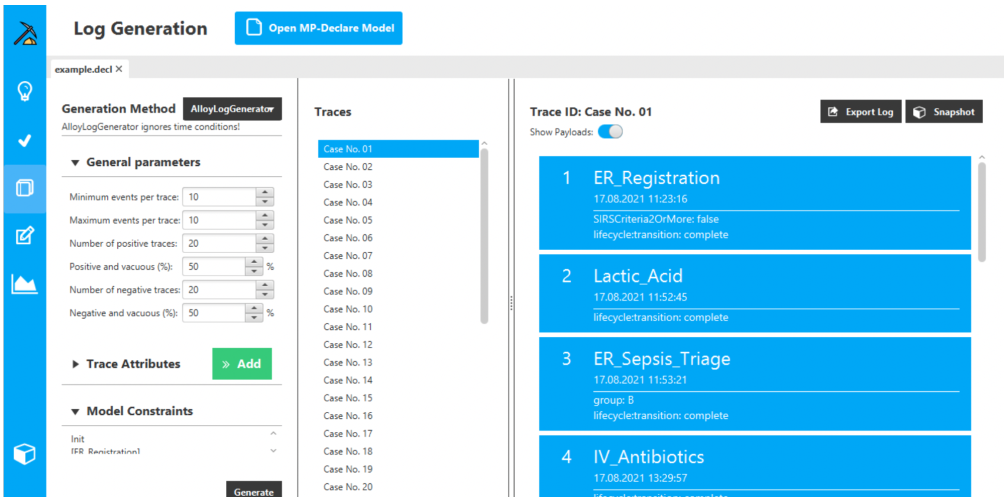
RuM also supports the generation of artificial event logs. The log generation supports the AlloyLogGenerator and the MINERful Log Generator methods, with the main difference being that the AlloyLogGenerator method can also account for data conditions in the input model [6]. Log generation can be used to test new process mining algorithms on pre-existing models or better understand how a process behaves.

The log generation results are presented in a single view divided into three panels. An example of this is given in Figure 4. The log itself is written in XES format and can be readily explored and analyzed upon generation.

3.1.4 Monitoring

RuM also provides the functionality of monitoring. It allows users to animate a Declare model by replaying a log over the model and showing if the model violates or satisfies

[7]Trace represents all the events that correspond to a given case



Generation Method: AlloyLogGenerator
AlloyLogGenerator ignores time conditions!

General parameters

Minimum events per trace:	10
Maximum events per trace:	10
Number of positive traces:	20
Positive and vacuous (%):	50 %
Number of negative traces:	20
Negative and vacuous (%):	50 %

Trace Attributes

Model Constraints

Init
[ER_Registration]

Traces

Case No. 01
Case No. 02
Case No. 03
Case No. 04
Case No. 05
Case No. 06
Case No. 07
Case No. 08
Case No. 09
Case No. 10
Case No. 11
Case No. 12
Case No. 13
Case No. 14
Case No. 15
Case No. 16
Case No. 17
Case No. 18
Case No. 19
Case No. 20

Trace ID: Case No. 01
Show Payloads:

1 ER_Registration
17.08.2021 11:23:16
SIRSCriteria2OrMore: false
lifecycle:transition: complete

2 Lactic_Acid
17.08.2021 11:52:45
lifecycle:transition: complete

3 ER_Sepsis_Triage
17.08.2021 11:53:21
group: B
lifecycle:transition: complete

4 IV_Antibiotics
17.08.2021 13:29:57

Figure 4. Log generation view.

the constraints defined in the model. This allows users to simulate the log behaviour over a Declare model and to observe and identify which specific constraints of the process model are temporarily or permanently satisfied or violated after each event in the event log. Monitoring supports three methods: MP-Declare w Alloy, MobuconLTL and MobuconLDL methods [6].

The monitoring results are presented in a single view divided into three panels. An example of this is given in Figure 5.

3.2 Declare Editor

The editor included in the currently public version of RuM supports standard Declare and MP-Declare (a Multi-Perspective version of Declare) [12]. With this feature, one can open and edit an existing Declare model or create a new model. The existing editor is presented in a single view, divided into two halves. The left panel can be used to edit activities and attributes. The model visualisation, constraints, and data conditions are found in the right panel. The model visualisation shows the graphical representation of the model, and model constraints can be added or modified on the constraint list in the right panel. The decl file format (discussed in more detail in Section 5.4) is used to export and open MP-Declare Editor. In the following sections, we will examine the editor's components and how they look and operate in the currently public version of RuM.

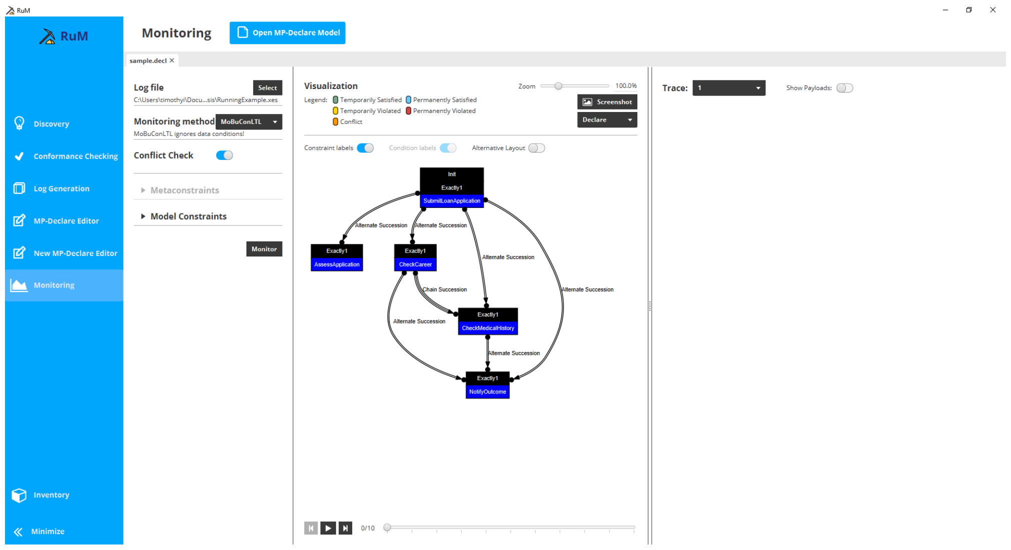


Figure 5. Monitoring view.

3.2.1 Activities

Activities are one of the main building blocks of the Declare modelling language. Each activity represents some specific task that is performed in a process. In the current Declare editor (current public version), a new activity can be added using the “Add activity” button in the left panel, under the “Activities & Attributes” section. Figure 6 shows the panel for adding new activities.

Clicking this button opens up the “Add activity” panel with two fields to fill:

- Activity Name: The activity name can be specified in this field.
- Attributes: The list of attributes associated with the activity can be specified in this field. If there are no attributes currently defined in the model, then the list of attributes will be empty.

The panel also contains two action buttons. The “Add activity” button to confirm the action, and the “Close” button to close the panel. Once added, the activities are displayed in a visualisation window in the right half of the editor view and in a list on the left panel with buttons to edit and delete them. The edit button (represented by the blue pencil icon) opens a similar panel with the activity name and attributes already filled and allows the modeller to edit the name and the attributes. The delete button (represented by the red waste bin icon) deletes the activity, removing it from the list of activities and from the visualisation window. Activities are represented as blue rectangular boxes in the visualisation window, with the name of the activity in the centre of these boxes.

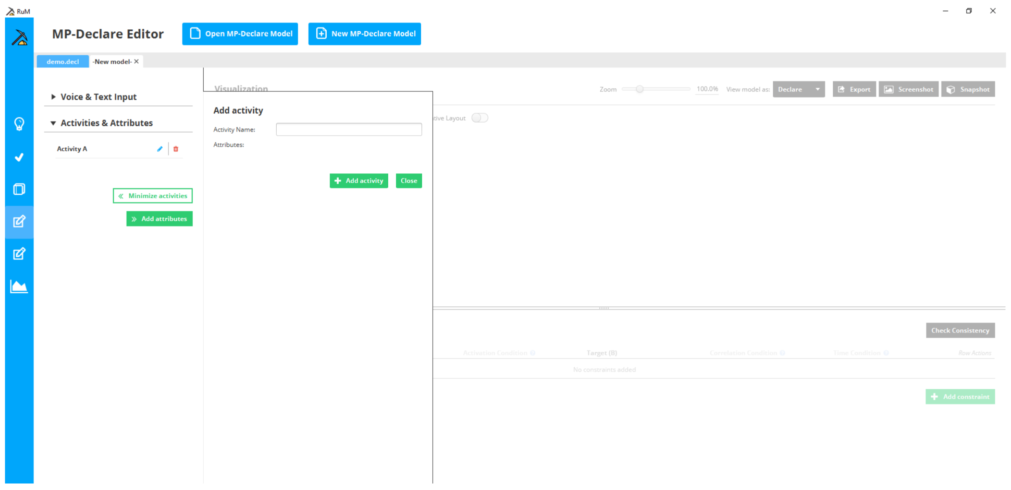


Figure 6. Add activity view.

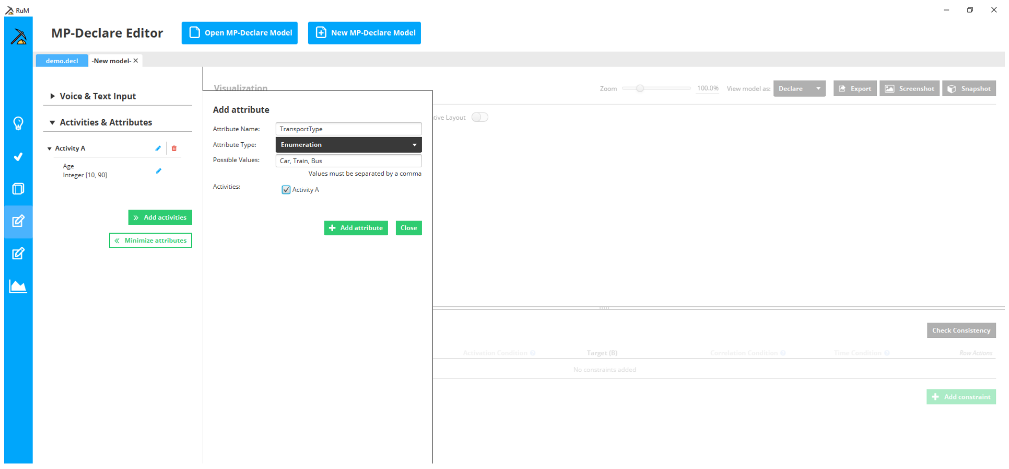
3.2.2 Attributes

Attributes are descriptors attached to an activity. In the editor, a new attribute can be added using the “Add attribute” button in the left panel, under the “Activities & Attributes” section. Figure 7 shows the panel for adding new attributes.

Clicking the “Add attribute” button opens up the corresponding panel with four fields to fill:

- Attribute Name: The attribute name can be specified in this field.
- Attribute Type: The type of the attribute can be selected from the dropdown. There are three options, integer, float and enumeration.
- Value Range/Possible Values: The data entered here depends on the selected attribute type. For the enumeration attribute type, a field to enter all possible values is presented, and for the integer or float type, two fields are shown to enter the range of values (i.e., a value from and a value to).
- Activities: The activities to be attached to this attribute are selected from the list of available activities in the model.

Similarly to the “Add activity” panel, the panel for adding attributes also contains two action buttons. The “Add attribute” button to confirm the action, and the “Close” button to close the panel. Once added, the attributes are displayed in a list in the left panel under the associated activities. Each attribute element contains the attribute name, type, value range and a button to edit the attribute. The edit button opens a similar panel to the create attribute panel with all the fields prefilled with the existing values. We note here that



Age
Integer [10, 90]

Attribute Name: TransportType
Attribute Type: Enumeration
Possible Values: Car, Train, Bus
Values must be separated by a comma
Activities: [✓ Activity A]

Figure 7. Add attribute view.

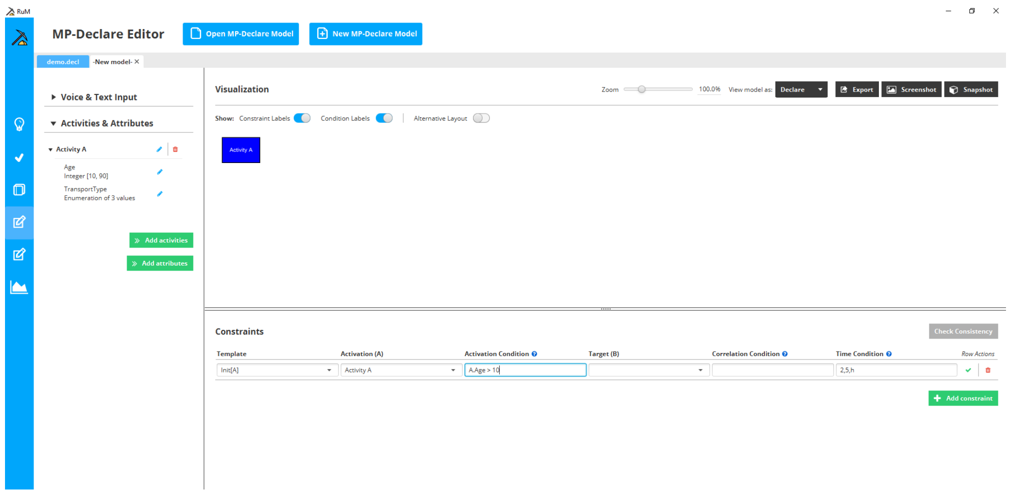
the attributes are visually represented only if they are used in some data conditions (see Section 3.2.4).

3.2.3 Constraints

Constraints form the backbone of the Declare language, with each constraint defining some important aspect of the process [3]. A constraint consists of a template and one or two activity references. The template, also known as the constraint type, defines the semantic meaning of the constraint. Templates are broadly divided into two groups, unary and binary constraints [2]; depending on the group a constraint belongs to, the number of activities referenced could be one or two, respectively.

In the editor, a new constraint can be added by using the "Add constraint" button, which can be found in the bottom half of the right panel. Figure 8 shows the panel for adding new constraints. Clicking on this button adds an empty row to the constraint table, which can also be found in the bottom half of the right panel. The table consists of the following columns:

- Template: The template or constraint type can be selected here.
- Activation (A): The source or initial activity is selected for the constraint type.
- Activation Condition: Conditions that need to be fulfilled for the activation to be triggered are specified in this field.
- Target (B): The target activity is selected for the constraint type.



Activity A
Age
Integer [10, 90]
TransportType
Enumeration of 3 values

Zoom 100.0% View model as: Declare

Show: Constraint Labels Condition Labels Alternative Layout

Constraints

Template	Activation (A)	Activation Condition	Target (B)	Correlation Condition	Time Condition	Row Actions
Init[A]	Activity A	A.Age > 10			2,5,h	✓ 🗑

Figure 8. Add constraint view.

- Correlation Condition: Conditions that correlate the target data with the source ones are specified in this field [19].

- Time Condition: The time conditions that need to be fulfilled are specified in this field.

Each row also contains two buttons to confirm the action and remove the added row, which corresponds to deleting the constraint contained in that row. Once added, the constraints are displayed in the constraint table. Each constraint also includes a button to edit and delete the constraint.

3.2.4 Data conditions

The standard Declare paradigm was extended to include conditions that predicate not only on activities and their control flow but also on data attributes and timestamps [12]. These conditions are called data conditions. They are three types of data conditions:

- Activation Condition: This expresses conditions that need to be fulfilled for the activation to be triggered.

- Correlation Condition: These conditions correlate the target data with the source ones.

- Time Condition: The time conditions express constraints over the time distance between the activation and the target of a constraint.

In the editor, these conditions can be specified on constraints depending on the type of constraint. Activation conditions and time conditions can be added to all the different groups of constraints (unary and binary constraints). Correlation conditions, on the other hand, can only be added to binary constraints.

There are rules associated with specifying each of the various types of data conditions. A valid time condition has the following syntax:

$$lower\_bound, upper\_bound, time\_unit \tag{1}$$

where:

$$\begin{array}{ll} lower\_bound & = \text{positive integer} \\ upper\_bound & = \text{positive integer} \\ time\_unit & = \text{one of the following: s, m, h or d, where } \begin{array}{l} s = \text{second} \\ m = \text{minute} \\ h = \text{hour} \\ d = \text{day.} \end{array} \end{array}$$

An example of a valid time condition is $2, 5, h$, which means that the distance between the activation and target should be at two hours and, at most, five hours. A valid (atomic) activation/correlation condition has the following syntax:

$$[attribute][operation][value(s)]. \tag{2}$$

The allowed atomic conditions depend on the specific attribute type. For enumeration attribute type, the following atomic conditions are supported:

- A.TransportType **is** Car
- T.TransportType **is not** Car
- T.TransportType **in** (Car, Train)
- A.TransportType **not in** (Car, Train)

For float and integer attribute types, the following atomic conditions are supported:

- A.age > 10
- A.age <= 5
- T.age = 3

where:

$A.$ = activation attribute
$T.$ = target attribute

The atomic conditions outlined above can also be joined with the logical operators "and" and "or", for example, $T.Age\ <=\ 10\, or\, T.Age\ >\ 70$.

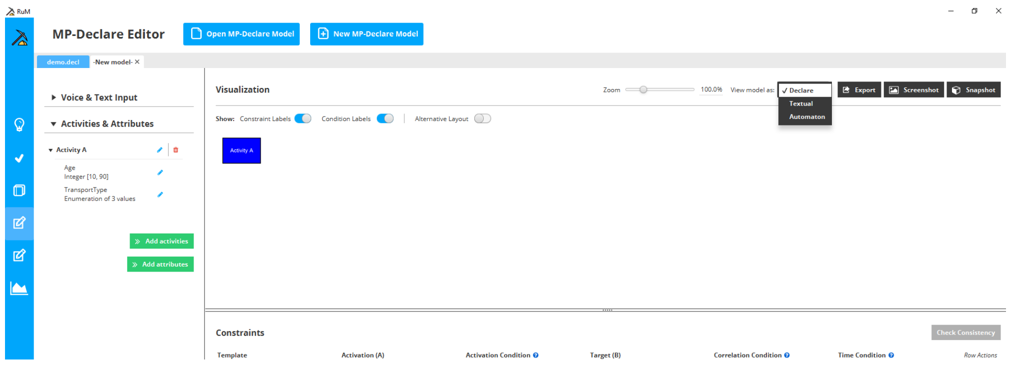


Figure 9. Visualization window.

3.2.5 Visualization window

The visualisation window is where the model is visually represented. The visualisation in the editor relies on the built-in browser support of JavaFX, which is combined with VisJs [30] to generate and display a graph in SVG format from a DOT file using Graphviz[8]. This DOT file is generated from scratch (each time the model is modified) by combining the various components (constraints, attributes, activities) and then displayed in the JavaFX browser element. One major disadvantage of this visualisation is that the graph is entirely static, meaning that the user has practically no control over where the components of the graph are located or how they are displayed.

There are three model views, namely Declare, Automaton and Textual. The default view is the Declare model view. The view can be toggled using the "View model as" dropdown at the top right of the visualisation pane. The visualisation also supports zooming in and out using a zoom bar. A screenshot can be taken of the visualisation, and it can also be exported. The only control that the user has over the visualisation is to use an alternative layout (which arranges elements from left-to-right instead of the default top-to-bottom layout) and show/hide constraint and/or condition labels.

An example of the visualisation window is shown in Figure 9.

3.3 Application Architecture

In this section, we describe the overall architecture of RuM. This section starts with an overview of the software design and then continues with an overview of the main technologies used in developing RuM. The section ends with a description of how the application is packaged.

[8]https://graphviz.org/

3.3.1 Software Design

RuM is developed following the MVC (Model-View-Controller) design pattern, which, as the name suggests, divides an application into three parts: model, view and controller [11]. The view object displays information and results to the user, observes the data model object for any changes and updates the information displayed to the user accordingly. The view object also gathers the user's input and passes it to the controller to perform actions. The model object encapsulates information, while the controller object implements the application's actions and logic. The controller acts as a middle link between the user and the model.

In RuM, the responsibility of these three parts can be further described as follows:

- The model object is responsible for the following:
  - The model object performs all process mining tasks the user requests. It actually runs the process mining algorithms.
  - The model object holds all data contained in the models and logs opened, created or modified by the user.
  - The model object performs all pre-processing and post-processing needed for a specific process mining task.
- The view object is responsible for the following:
  - The view object receives all inputs from the user.
  - The view object displays the results of a processing task and allows users to navigate through these results.
  - The view object provides components and elements to facilitate the user's interactions with the application. For example, it provides buttons for navigation and input fields.
  - The view object also displays notifications and the current status of the application. For example, notifications of successful processing or failure of a certain task.
- The controller object is responsible for the following:
  - The controller object receives all inputs from the user, validates them and processes them.
  - The controller object handles navigation requests from the user and changes the view in the user interface.
  - The controller object loads data contained in the models and logs.

  - The controller object prepares and starts all process mining tasks the user requests.
  - The controller object also prepares the results from all process mining tasks to be displayed in the view.

All application views and user interface components in RuM are in FXML files. FXML[9] is a scriptable, XML-based markup language for constructing Java object graphs. It provides a convenient alternative to constructing such graphs in procedural code and is ideally suited to defining the user interface of a JavaFX application since the hierarchical structure of an XML document closely parallels the structure of the JavaFX scene graph [20].

3.3.2 Technology Used

The current version of RuM is written in Java 11, which is the Long-Term-Support (LTS) version of Java that receives security updates at least until 2026. JavaFX is used as a user interface framework, and its companion tool, Scene Builder, is used for designing application views. The model visualizations in RuM rely on the built-in browser support of JavaFX. This inbuilt browser is used together with VizJS to generate and display graphs from a DOT file to SVG using Graphviz. This approach is also used in the current publicly available editor. Atlassian's Bitbucket is used for source control. Bitbucket is used to track and manage changes in the code base. In this way, every developer and user of the application has the exact version of the application at any given time and can, if needed, work independently from other developers.

3.3.3 Application Packaging

RuM is currently compiled into a single runnable JAR package. Apache Maven Shade Plugin is used to package RuM together with all of its dependencies. This means that RuM can be started directly from JAR package, with the only requirement being the installation of Java 11 JDK. The compiled JAR package also contains a specific JavaFX library for all operating systems. The main advantage of this is that users can use the same JAR file on various operating systems (Windows, Linux and Mac). They can run this RuM Package via the command line (java -jar .\rum-0.6.6.jar) or, if `JAVA_HOME` is set, simply by double-clicking the JAR file.

3.4 Related Work

In this section, we will discuss some related work concerning RuM, which is the focus of this thesis. In 2019, the first version of RuM was proposed and developed in the master's

[9]https://openjfx.io/javadoc/15/javafx.fxml/javafx/fxml/doc-files/introduction_to_fxml.html

thesis of D. Kapisiz [17]. This tool was developed to create the first comprehensive desktop application for declarative process mining with an intuitive user interface. However, it followed quite basic and old-fashioned approach to user interface design.

In 2020, the second version of RuM was developed in the master's thesis of A. Alman [2]. This version of RuM was developed to improve the previous version, improve the user interface, and make it faster and more user-friendly.

In [6], A. Alman et al. discuss how the 2020 version of RuM works, the design goals, and present a user evaluation and a summary of possible improvements, including the development of a visual editor for Declare models.

The current publicly available version of RuM is based on the aforementioned works, however, it has been further improved and extended over the past few years by adding new functionalities, algorithms, etc. For example, the monitoring functionality was added in [5].

Also, in 2020, A. Alman et al. presented a chatbot, Declo [4], that allows modellers to define constraints using speech (i.e. natural language sentences) in vocal or textual form.

Furthermore, in 2021, a tutorial about RuM [7] was presented at the International Conference on Business Process Management.

4 Method

This thesis work aims to develop a visual Declare editor for the RuM toolkit. We decided to apply the Design Science Research (DSR) methodology [25] to do this. In DSR, the researcher combines the theoretical result with their practical application by developing an artefact. In this way, the research becomes useful for academic and industrial audiences. The design science research has been widely adopted in engineering disciplines as a way to frame research by highlighting both the problems addressed and the interventions proposed while supporting researchers in conveying how they build on and contribute to an existing knowledge base [25]. It involves the design and the development of artifacts for these purposes. The artifacts we study are designed to interact with a problem context[10] to improve something in that context [31].

DSR starts with the problem and context definition. To develop our proposed solution, we needed to gather information (user requirements) on the problem context. To do so, we (1) analyzed the RuM user evaluation results from [2], and (2) the feedback received by the users over the past few years. The results of this analysis revealed a development opportunity, which served as the basis for this thesis. After defining the problem context, the next step is the design cycle. In the design cycle, we design and develop the new artefact. Finally, the artefact is evaluated by experts in the domain in order to verify whether it produces the desired effects and whether a new iteration over the design cycle is needed.

The chapter continues with a description of how the requirement elicitation was conducted, followed by a detailed overview of the proposed steps in the design cycle. Lastly, the chapter concludes with a description of the method used in the evaluation process to verify the newly developed artefact.

4.1 Requirement Elicitation

This is the process of gathering a list of requirements from various stakeholders (usually the users) to be used as the basis for the formal requirement definition. In the thesis work of A. Alman, in 2020, a user evaluation was done on the version of RuM developed within that thesis [2]. As the development of the Declare editor, since that user evaluation, has been limited to bugfixes and no major revisions of the user interface have occurred, it formed a natural input for the work in this thesis. The results of that user evaluation, combined with user feedback and suggestions coming from researchers using the application over the past few years, formed the basis for this thesis.

In A. Alman's thesis, eight participants from various European universities were selected for the evaluation. The participants were all experts in process mining. Four participants had little to no knowledge about the Declare process modelling language,

[10]Problem context refers to an identified challenge or issue to be solved.

while the other four were Declare experts. This split was done to test whether the application was useful for both experts and users unfamiliar with the Declare language. The participants were given a curated set of tasks to carry out while being recorded. These tasks covered all parts of RuM, including the Declare editor. After completing the tasks, a short interview was done with the participants to further elicit concerns and critical points. After this, a survey was also sent out to the participants to capture anything not previously captured from the initial processes. Some of the major critical points with the Declare editor raised during the user evaluation were the inability to move activities around in the visualisation window, the inability to perform basic actions with the mouse like panning and stretching the model, the lack of proper validation for user inputs, the inability to draw a model in the way the user wanted (the system fully defined the layout of the model), availability of only a table-based model editor, poor error handling. These critical points from all the participants were aggregated to form the requirements which form the basis of this thesis.

With the requirements now available, we needed to prioritise them. In this thesis, we use the MoSCoW technique [26], which is a four-step prioritisation technique where the requirements are categorised according to the importance of delivering each requirement. There are four priority groups: MUST have, SHOULD have, COULD have, and WON'T have.

- MUST have - requirements that are mandatory.
- SHOULD have - requirements that are of high priority but not mandatory.
- COULD have - requirements that are preferred but not necessary.
- WON'T have - requirements that are irrelevant and can be postponed.

Tasks are generated from the requirements and propagated to the project board for execution. The requirements of the new editor, along with their prioritisation, are described in Section 5.1 below.

4.2 Design Cycle

In this cycle, the new editor is designed and developed. Here, we adopt the Agile Methodology, which is an iterative approach to software development and project management that helps teams deliver value to their customers and users faster and with fewer issues [8]. This methodology was chosen because of its flexibility which provides the ability to respond to changes. With how fast software libraries/packages release updates and how the user's wants and needs constantly change, we needed a framework that permits and encourages responding to these changes rapidly. We also wanted a framework that prioritises the users over the development process, and the Agile manifesto [18] supports this. More specifically, the agile manifesto for software development places value on the following:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

The design cycle involves two parts: design and development.

4.2.1 Design and Development

The design goal within the context of this thesis is to have a visual editor with easily navigable, simple-to-learn minimalist design. With this goal in mind, mock-ups of the design were created. A mock-up is a static design of a web page or application that is not functional but shows many of the features that will be contained in the final product. This serves as a visual draft for the application, allowing stakeholders and developers to agree on what the application or page should look like and suggest layout, colour and style changes. The agreed mock-ups served as the basis for developing the new editor.

The mock-ups were created using a tool called Figma[11]. Figma was selected because it is a robust, widely used, free tool for (user interface) design. It is a collaborative web application which can be easily used to share semi-interactive[12] mock-ups of a user interface. Once the design aspect is done and agreed upon, the next phase in a cycle is development.

Before the design and the development cycles could begin, we also need to identify how we will build the new editor, and what technology stack will be used. Because we wanted to build a graphical editor for Declare models (which can be seen as graphs, where nodes represent activities and lines represent relations between them), we highlighted a few essential requirements in terms of the elements we need to be able to represent and allow the user to interact with. The technology selection was based on both the feasibility and difficulty of creating and implementing these requirements. The main requirements were the following:

- Ability to draw and customise a node (i.e., an activity) into any shape and style.
- Ability to create an interactive visualisation window (i.e., a visualisation window that allows users to move components around and allows for basic manipulation like panning and zooming).

[11] https://www.figma.com/

[12] Figma allows, for example, to navigate between different static but interconnected mock-ups, which can be used to create an illusion of limited interactivity.

- Ability to layout an imported Declare model properly in the visualisation window.
- Ability to create double and triple line style (i.e., a connection line between two activities with double or triple parallel lines to represent binary constraints, specifically constraints of type alternate and chain).
- Ability to create dashed lines (including for double and triple lines), as dashed lines are used to represent negation of binary constraints.
- Ability to draw multiple lines between two nodes (i.e., multiple binary constraints related to exactly the same activities).
- Ability to create composite nodes (i.e., an activity and one or more unary constraints).
- Ability to have lines with various tip endings, e.g., arrows, circles, etc., as these shapes are used to visually differentiate between different binary constraints.

The details about the technologies explored, the process of selecting a technology stack and the selected technology stack are detailed in Section 5.2.

4.3 User Evaluation Methodology

To evaluate the new editor, a qualitative user evaluation was conducted, with the focus fully on the new editor. Users with varying competence with the Declare notation were selected for this evaluation. Participants, who are all process mining experts with academic backgrounds, were selected for the study from various universities across Europe. Various scenarios were highlighted and presented to each user to complete while we recorded the actions. The participants were encouraged to think out aloud, give suggestions, and ask questions (e.g., when it was unclear how to solve some task or the meaning of some button/field was unclear). The principal investigator asked additional questions to understand the participants thinking when solving each scenario. At the end of the evaluation, we had a quick interview to review some of the challenges or suggestions to confirm that there were no misunderstandings. Finally, an anonymous questionnaire was sent out to the participants to also gather their thoughts afterwards. The questionnaire was used to evaluate the new editor.

We evaluated the new editor using the following metrics:

- Usability [27] - How usable was the new editor?
- Satisfaction [10] - How satisfied is the participant with the new editor?
- Expectation Fulfilment [10] - Were there any feature shortcomings from the participant's point of view

- Future use [10] - How likely is it that the participant would use this editor again?
- Usefulness [10] - Would this editor be a useful alternative to the previous way of creating a model?

The usability questions were formulated using the System Usability Scale (SUS) proposed by John Brooke in [27]. All the above metrics except "Usability" were scored from one to ten (1 - 10), with one being the worst case and ten the best case. The usability was scored from one to hundred (1 - 100), where the minimum passable score is considered to be 68 according to the SUS standard [27].

The results of the user evaluation will be presented in the last part of this thesis.

5 The New Graphical Declare Editor

In this section, we introduce the main contribution of this thesis. We start by highlighting the results of the requirement collection, followed by a discussion on the technology stack used to build the new graphical editor. We then provide a detailed description of the new graphical editor and its components.

The source code of the application is available in a public Bitbucket repository: `https://bitbucket.org/doorless1634/thesis/src/new_editor/`. The compiled version of the application can be found in a publicly viewable Google Drive folder: `https://drive.google.com/drive/folders/1yogJyVAjnWOYd-iVS1Q4sA0JmUhfURbq`.

5.1 Requirements

This section highlights the results of the requirement collection and the prioritisation of these requirements.

5.1.1 Functional Requirements

As mentioned in Section 4.1, functional requirements were extracted from the user evaluation done in the A. Alman's thesis in 2020 and the user feedback collected over the years. These requirements are presented in Table 4. Each requirement line has a unique identifier and a description.

Table 4. Functional requirements.

ID	Description
FR-1	Ability to create a new activity
FR-2	Ability update an activity name
FR-3	Ability to delete an activity
FR-4	Ability to create an attribute and attach it to activities
FR-5	Ability to update an attribute and also to remove it from activities
FR-6	Ability to delete attribute
FR-7	Ability to add a unary constraint to an activity
FR-8	Ability to remove a unary constraint from an activity
FR-9	Ability to add binary constraint between two activities
FR-10	Ability to change the type (template) of a binary constraint
FR-11	Ability to delete a binary constraint between two activities
FR-12	Ability to add data conditions to constraints
FR-13	Ability to edit data conditions on constraints
FR-14	Ability to remove data conditions from constraints

FR-15	Ability to validate and enforce data condition rules to ensure accurate and consistent data conditions
FR-16	The editor should have an interactive visualisation window for the model
FR-17	The editor should have an Automaton view
FR-18	The editor should have a Textual view
FR-19	Ability to import an existing Declare model and lay it out properly
FR-20	Ability to export a model as a Declare model, as an Automaton and as a Text file
FR-21	Ability to import and automatically layout a Declare model created with the previous version of the editor
FR-22	Ability to produce screenshots of the Declare, Automaton and Textual views of the model

5.1.2 Non-Functional Requirements

In Table 5, the non-functional requirements are presented. Similar to the functional requirements, each requirement line has a unique identifier and a description.

Table 5. Non-functional requirements.

ID	**Description**
NFR-1	The editor should have a user-friendly interface that requires minimal training
NFR-2	The editor's code design should follow industry-standard best practices
NFR-3	The editor should be reliable, trustworthy, and consistent

5.1.3 Requirement Prioritisation

To prioritise the requirements, we use the MosCoW prioritisation model, as mentioned in Section 4.1. Table 6 shows how the requirements are grouped according to the priorities.

Table 6. Requirements prioritisation.

Priority	**Requirements**
Must have	FR-1, FR-3, FR-4, FR-6, FR-7, FR-8, FR-9, FR-11, FR-12, FR-14, FR-16
Should have	FR-2, FR-5, FR-10, FR-13, FR-15, FR-20, FR-21
Could have	FR-17, FR 18, FR-19, FR-22

5.2 Technology Stack

To select the technology stack for the development of the new editor, we identified the following three options:

- Building the editor using a Javascript library called Diagram-js[13].
- Building the editor using a Javascript library called Cytoscape[14].
- Building the editor using a Java library called JavaFX[15].

The first two options involve (at least partially) changing the language used for the editor from Java to Javascript, while the last option is based on the same language used in other parts of the RuM toolkit. We considered that changing the language to Javascript would have increased the speed and user experience because of its versatility and interactive nature in building front-facing applications, and at first glance, both Diagram-js and Cytoscape seemed to natively support some of the features we required. Meanwhile using JavaFX would have allowed for more flexibility (but also more development complexity) because all visual components and much of the functionalities would have be built from the ground up with the Declare notation in mind.

To decide what technology to use, example projects were created using the three options identified above and evaluated using the metrics highlighted in Section 4.2.1.

Diagram-js was created by a small group of developers focusing on BPMN (Business Process Modelling Notation) 2.0 diagramming. Three libraries have been built on top of diagram-js:

- bpmn-js: A modeller and viewer for BPMN 2.0
- cmmn-js: A modeller and viewer for CMMN (Case Management Model and Notation) 1.1
- dmn-js: A modeller, viewer and table editor for DMN (Decision Model Notation) 1.3

Seeing that these libraries were built using diagram-js, we investigated whether we could also build a library to support the Declare notation in a similar fashion and then use this library to build a graphical editor for RuM. We created an example project and tried creating some custom components for Declare. We could draw nodes of different shapes (triangle, square and circle nodes), and there was an interactive visualisation window available. An example of this can be seen in Figure 10. However, we could not create

[13]https://www.npmjs.com/package/diagram-js
[14]https://www.npmjs.com/package/cytoscape
[15]https://openjfx.io/

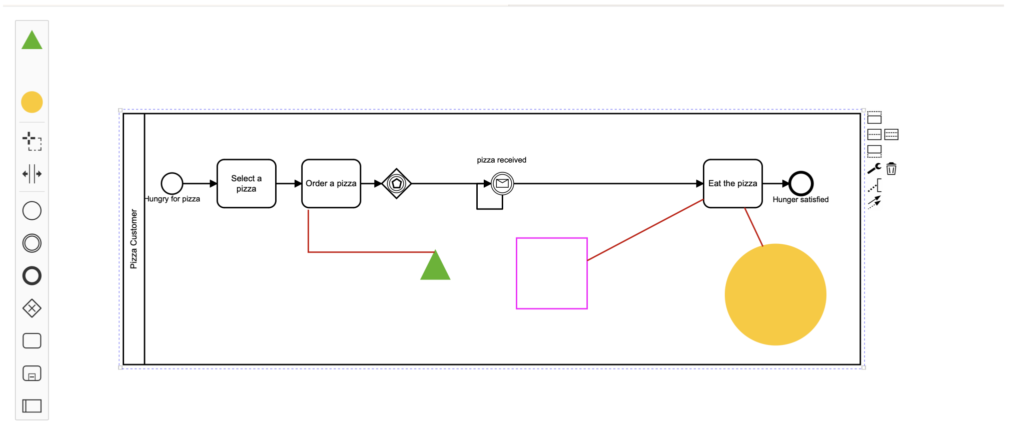


Figure 10. Different node types created in the diagram-js example project.

double and triple line styles (necessary for representing alternate and chain constraints), we could not create composite nodes (for representing unary constraints), and the custom nodes (triangle, square and circle nodes) created had a separate rule for the kind of lines. We concluded that this library was built for BPMN (Business Process Modelling and Notation) and was not flexible enough to be adapted to the Declare notation without spending considerable effort on rewriting the core components of the library.

Cytoscape, while not originally intended for process models, it is successfully used, for example, in Apromore, to show process discovery results (in the form of graphs). In that case, Cytoscape also handles the layouting of the process discovery results automatically, based only on the names of the nodes and the connections between them. Figure 11 shows the automatic layout in Cytoscape. However, Cytoscape lacks built-in support for most of the features required for a model editor (users can move around the nodes, but most other features would need to be implemented during the development of the new Declare editor). The implementation effort to create these required features would be comparable to implementing everything with JavaFX, and using Cytoscape would also mean adding a new programming language (Javascript) to the overall technology stack of RuM while providing a clear benefit only for the automatic layout of models.

Technically it might be possible to create all the components and elements needed to build the new editor with the first two options, however, the time and effort required would be more or at least comparable to using JavaFX. Also, the first two options would mean adding a new programming language to RuM's technology stack, which presents a new overhead to the application. Because of these reasons, the last option (using JavaFX) was selected. The challenge with this option was that it required building the editor and its components from the ground up. In the subsequent sections, we will detail these components and how they were built.

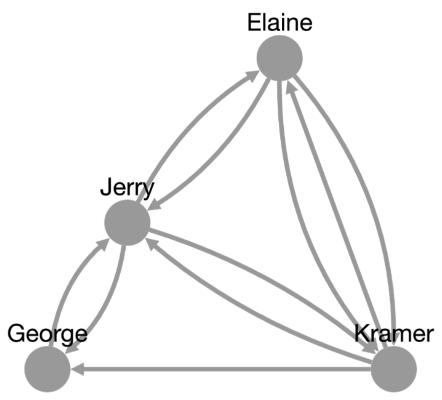


Figure 11. Node and edges layout in Cytoscape.

JavaFX is an open-source, next-generation client application platform for desktop, embedded systems and mobile built on Java [16]. The platform is composed of a set of graphics and media packages that enable developers to design, create, test, debug, and deploy rich and robust client applications that operate consistently across diverse platforms [21]. JavaFX also allows for the use of CSS in styling the components.

5.3 Core Application

In this section, we introduce the new Declare editor, the editor's components, and the technical details about all the aspects of the editor. We start by introducing a class diagram describing the structure of the core classes in the editor shown in Figure 12, which is followed by a description of the user interface of the new editor.

The ActivityNode class represents an activity, and it is created using an ActivitySkeleton. An ActivitySkeleton, as the name implies, serves as the building block to create an activity. It contains details like the id, the name of the activity and the initial position coordinates (i.e., the x and y coordinates of the activity in the process map).

The Connection class represents a line or a connection created between two activities (representing a binary constraint). It contains details such as the source activity, the target activity and the constraint type. It also allows specifying the particular connector points on the activities (i.e., where exactly the line connects to the border of the activity nodes)

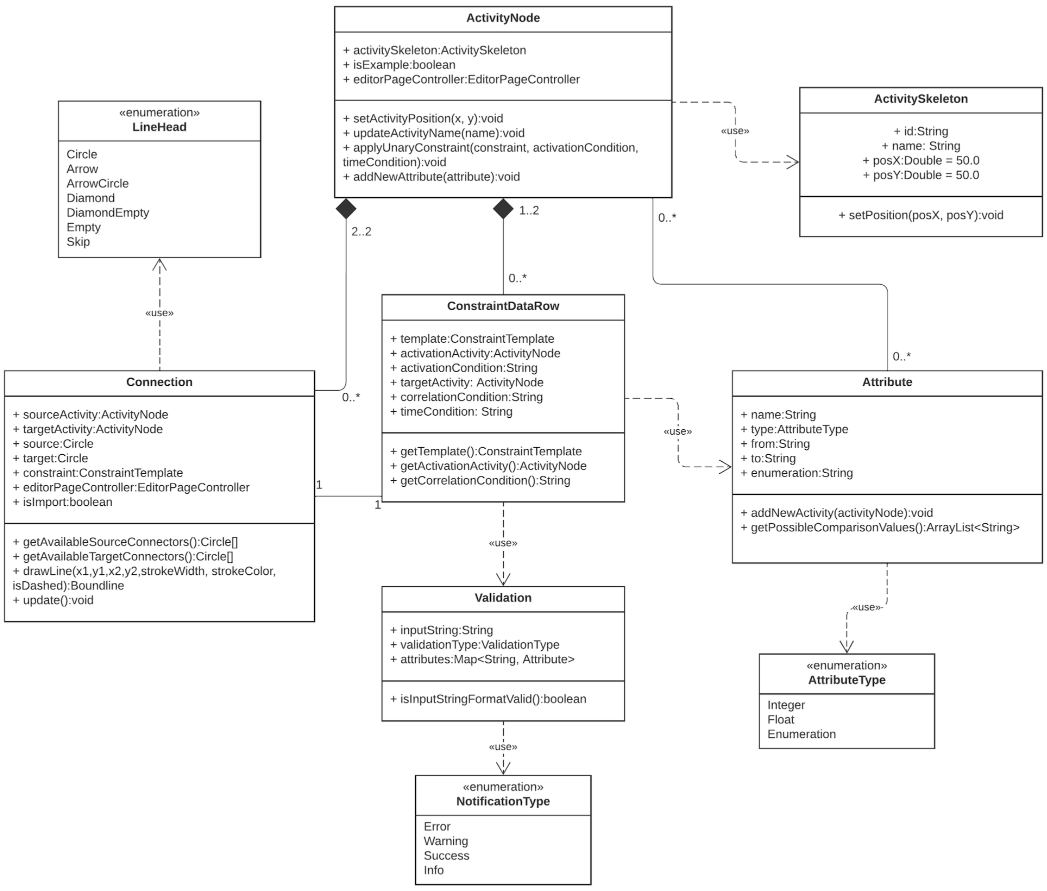

ActivityNode
+ activitySkeleton:ActivitySkeleton
+ isExample:boolean
+ editorPageController:EditorPageController
+ setActivityPosition(x, y):void
+ updateActivityName(name):void
+ applyUnaryConstraint(constraint, activationCondition, timeCondition):void
+ addNewAttribute(attribute):void
«use»
ActivitySkeleton
+ id:String
+ name: String
+ posX:Double = 50.0
+ posY:Double = 50.0
+ setPosition(posX, posY):void
«enumeration»
LineHead
Circle
Arrow
ArrowCircle
Diamond
DiamondEmpty
Empty
Skip
2..2
1..2
0..*
0..*
«use»
ConstraintDataRow
+ template:ConstraintTemplate
+ activationActivity:ActivityNode
+ activationCondition:String
+ targetActivity: ActivityNode
+ correlationCondition:String
+ timeCondition: String
+ getTemplate():ConstraintTemplate
+ getActivationActivity():ActivityNode
+ getCorrelationCondition():String
0..*
Connection
+ sourceActivity:ActivityNode
+ targetActivity:ActivityNode
+ source:Circle
+ target:Circle
+ constraint:ConstraintTemplate
+ editorPageController:EditorPageController
+ isImport:boolean
+ getAvailableSourceConnectors():Circle[]
+ getAvailableTargetConnectors():Circle[]
+ drawLine(x1,y1,x2,y2,strokeWidth, strokeColor, isDashed):Boundline
+ update():void
0..*
1
1
«use»
Attribute
+ name:String
+ type:AttributeType
+ from:String
+ to:String
+ enumeration:String
+ addNewActivity(activityNode):void
+ getPossibleComparisonValues():ArrayList<String>
«use»
Validation
+ inputString:String
+ validationType:ValidationType
+ attributes:Map<String, Attribute>
+ isInputStringFormatValid():boolean
«use»
«enumeration»
AttributeType
Integer
Float
Enumeration
«use»
«enumeration»
NotificationType
Error
Warning
Success
Info

Figure 12. Class diagram of the editor.

using the 'target' and 'source' fields.

The Attribute class represents the attribute associated with an activity. An attribute can be specified using details like name, from, to, enumeration and type. The fields 'from' and 'to' are used if the attribute type is 'Integer' or 'Float'. The field 'enumeration' is used if the attribute type is 'Enumeration'. Thus, the usage of the fields is enforced by the business logic of the application.

The ConstraintDataRow class represents a constraint data row. Each row has a constraint template, represented by the field 'template', and an activation activity, represented by the field 'activationActivity'. Activation condition and time condition (represented by fields 'activationCondition' and 'timeCondition', respectively) are optional for all constraints. Furthermore, if the constraint template is a binary one, the target activity (field 'targetActivity') is mandatory. Finally, the correlation condition (field 'correlation-

Condition') is optional, but it may only be filled for binary constraints.

The ConstraintDataRow class uses the Validation class to check and validate each data row input. The validation class validates and returns the validation result as a boolean value. The class also notifies the user when the input is invalid by displaying a suitable notification, which can be one of the four types defined in the enumeration 'NotificationType'.

To fulfil the aim of building a graphical easy-to-use editor, new user interface and model visualisation components were built from the ground up. The stylistic direction of the new editor is loosely based on modern editors and graphical design tools like Figma while still maintaining the colour definition of the RuM application itself. The editor is built like a single-page application (i.e., it is built on a single page and interacts with the user by dynamically rewriting and updating that page with new data). The following sections will look at the user interface components in detail, describing how they look and operate and shed light on the technical details behind them.

5.3.1 Control Palette

The Control Palette is a new component introduced into the model editor. As its name implies, it houses all the control actions used in the design of a new model. It is located on the left side of the editor, and it is divided into four sections: General, Unary Constraints, Binary Constraints and Attributes. Each of these sections is built as a collapsible panel (i.e., the user can show/hide the sections that the user is currently working with or is currently not using).

The General section contains two controls, a hand and an activity. The hand icon is a toggleable tool; when active, it allows the canvas to be moved/paned from left to right or top to bottom. The activity icon allows for the creation of an activity by dragging and dropping it into the canvas (discussed in more detail in the last part of this chapter). The sections for Unary and Binary constraints contain, respectively, controls for each unary and binary constraint available in RuM. The Attribute section initially contains just the button "Add attribute" for adding an attribute, and then when attributes are available, it also contains the list of these attributes. The Control Palette is shown on the left side (labelled with'1') of Figure 13.

5.3.2 Canvas

The canvas is on the right side of the Control Palette and is shown on the right side (labelled with '2') of Figure 13. It is the primary area for interacting with the model being created or modified. The canvas is initially a blank white area when a new model is created. Items are created mainly by dragging them from the Control Palette and dropping them into the canvas. The canvas can be zoomed in or out by using the mouse's scroll wheel or the pinch and spread gesture on a track-pad. The canvas also provides the ability

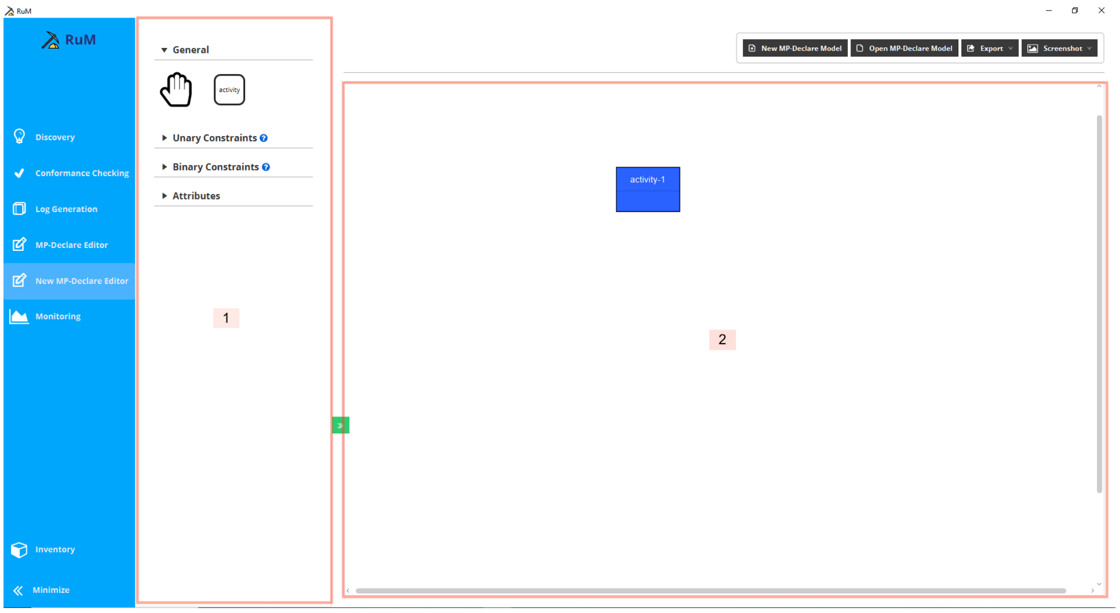


Figure 13. Editor showing the Control Palette (1) and the canvas (2) containing one activity.

to pan around to observe specific areas of the model a bit more closely. To pan around, the hand icon has to be active. The hand can be activated by clicking on the hand icon and then deactivated by clicking again on the same icon. When active, the hand icon is greyed out.

Model components like activities or connections can be moved around in the canvas by dragging and dropping them. Behind the scenes, we created an algorithm to calculate the actual position for a drop, factoring in the user mouse position, the canvas position, size and zoom level. The canvas also provides constant information about the position of the various components in the canvas with respect to other components, thus allowing the overall visualisation to be easily updated.

5.3.3 Activities

The visual representation of activities has been redesigned. Although still represented by a blue rectangular box, this box has been modified to look more modern by using a more suitable shade of blue, adding a softer border, a border-radius, and some shadow. An activity also has 16 connectors (small circles) around it, which are only visible when hovering over an activity. These connectors are used to connect binary constraints to activities (further discussed in the following sections). An example of an activity in shown in the canvas in Figure 13.

In the new editor, an activity can be added by dragging the activity from "General" section of the Control Palette. A simple drag-and-drop gesture from the Control Palette to the editor canvas creates a new activity with a default name which is the combination of the label "activity" and the count of activities already created. For example, the default activity name for the first activity created in the canvas is "activity-1". This simple drag-and-drop gesture reduces the previous amount of steps to create an activity to one. The activity name can be edited by double-clicking on it. The activity name becomes editable and, after editing, the new name is saved by pressing the "enter" key or by clicking anywhere outside the activity box.

An activity box is divided into three parts: the unary constraints container, the activity name container and the attribute container (Figure 14). The unary constraints container houses all the unary constraints currently attached to the activity, and it is the topmost container in the activity box. Below the unary constraint container there is the container for the activity name. And the bottom-most part of the activity box is the attribute container, which houses all the attributes attached to the activity. The activity also has various actions connected to the right and left mouse click. The left click of an activity shows a properties panel, which includes the details about the activity, like the name of the activity (which can also be edited in the properties panel), a table containing the attributes associated with the activity, and a delete button to delete the activity. The right click of an activity shows a context menu which contains buttons to add unary constraints to the activity.

When an activity is created, a couple of things happen in the background technically. When the activity is dragged to the canvas, the position information is used to create an activity skeleton which in turn is used to create an activity node. Then, the activity connectors are created and evenly spread and attached to the activity, and the visualisation is updated with this newly created activity.

5.3.4 Constraints

Constraints are the primary building blocks of the Declare language, and each constraint defines some important aspects of the business process being modelled. Constraints can be divided into two types based on how many activities they refer to. The two types are unary constraints and binary constraints. Unary constraints refer to one activity, while binary constraints refer to two activities. In the new editor, this division was used to better structure how the available templates are presented to the user.

There are two ways to add both constraint types to an activity or set of activities. To add a unary constraint, one option is to right-click on the activity. In this case, a popup will appear on the activity with all the available unary constraints, and the user can then select the desired constraint. A unary constraint can also be added to an activity by dragging it from the Control Palette onto an activity in the canvas. In this case, the first step is to open the "Unary Constraints" section in the Control Palette and then drag

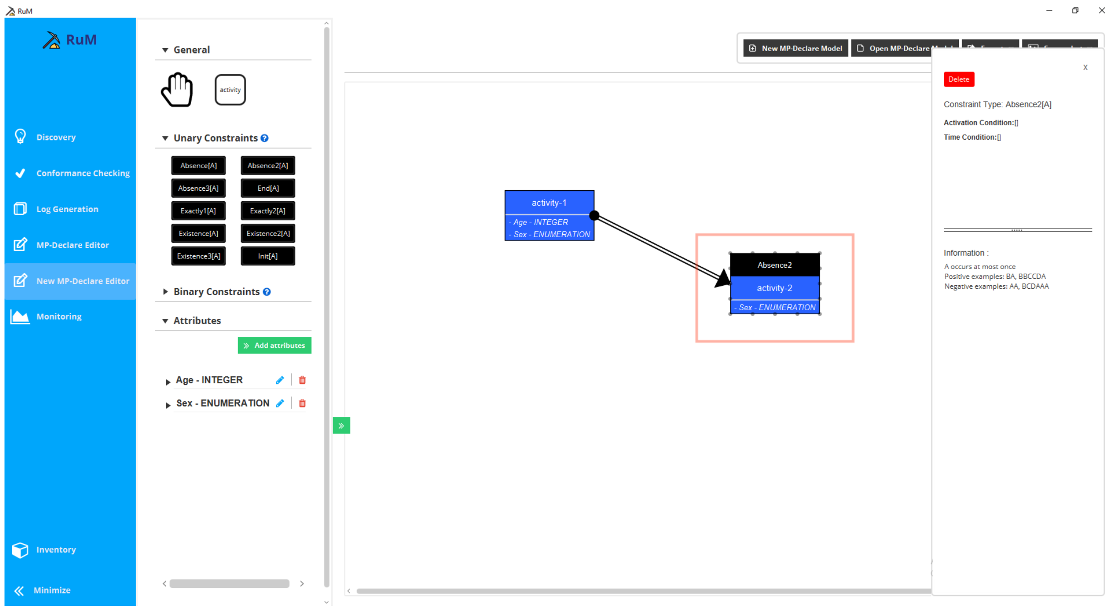


Figure 14. Editor showing a unary constraint.

and drop the desired unary constraint onto an activity. When a unary constraint is added, a new line entry (using the constraint template name) is added into the unary constraints container in the activity box.

Left-Clicking on an entry opens up a properties panel on the right side with details about the unary constraint like the template name, activation condition, time condition and an example explaining the template. Figure 14 shows an activity with a unary constraint and the unary constraint properties panel opened. In the properties panel, the user can edit the activation condition and the time condition, and also delete the unary constraint. Right-Clicking on the unary constraint on the activity in the canvas opens up a context menu with a delete button that can be used to delete the unary constraint.

Because the binary constraints involve two activities, the way to add them is slightly different. The first option involves dragging a binary constraint from the Control Palette onto an activity in the canvas and then selecting the second activity. In this case, the first step is to open the “Binary Constraints” section in the Control Palette, then to drag and drop the desired binary constraint template onto the activation activity (this will highlight all the other activities in the model), and finally to select the desired target activity from the highlighted ones. This will draw the line representing the binary constraint between the initial activity (activation activity) and the target activity. The specific connectors that the lines are attached to for both the activation and target activity are determined by the editor automatically by calculating the closest available connectors to activation and target activities.

The second option to add a binary constraint to a model involves drawing the connection line from the activation activity to the target activity on the canvas. To do this, the user needs to click and drag from one of the connectors of the activation activity (creating an initial line from the activation activity) and then drop on one of the connectors of the target activity (connecting the initial line to the target activity). When the drag motion is started, a line appears and follows the mouse until it is dropped on a target activity. After this drag-and-drop motion, a default connection line (which can be later changed into any binary constraint type) is created.

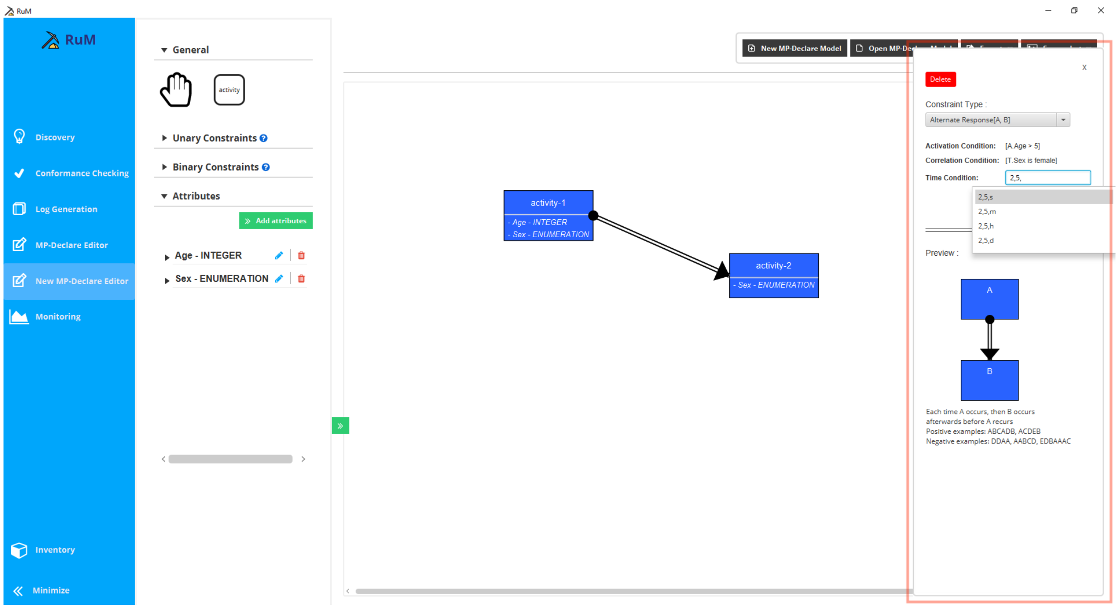
The line drawn when a binary constraint is created is a connection. Figure 15 shows a binary constraint 'Alternate Response' between "activity 1" and "activity 2" in the canvas. This connection, like the unary constraint container, has a couple of controls attached to the left and right mouse click. Left-clicking a connection opens a properties panel on the right side with details about the binary constraint like the template name, activation condition, correlation condition, time condition and an example explaining the template. The connection can also be changed or deleted in the properties panel. Right-clicking on the connection opens up a context menu with a list of binary constraint templates, allowing the modeller to change the template type, and a delete button, allowing the modeller to delete the connection. The connection can also be broken into two lines connected by a mid-connector to allow users to change the angle of the line. To break the line, the modeller has to drag the connection line from any part, and this breaks the line automatically into two at the point where the drag was initiated.

From the technical point of view, when a constraint is created, unary or binary, we also create a constraint data row. The Constraint data row consists of the template name and data conditions. In Section 5.3.5, we will discuss further the data conditions.

5.3.5 Data Conditions

In the editor, for a constraint, three types of data conditions can be specified: activation conditions, correlation conditions and time conditions. There are rules associated with each of these data condition types (as detailed in Section 3.2.4). One of the shortcomings of the old editor was an almost complete lack of validation of the data conditions. This shortcoming was addressed in the new editor by creating validators for each data condition type. Furthermore, the created validators also compare the entered data conditions to the attribute names and types in the model to highlight any mismatches. In addition to validating data conditions, the validation class also notifies the users via helpful error messages when the conditions are invalid.

Each data condition (regardless of the type) is always specified for a specific constraint. As stated earlier, when a constraint is created, a constraint data row instance is created, and this instance also contains the data conditions of the corresponding constraint. The data conditions available for various types of constraints differ. Activation and time conditions can be specified on all types of constraints (unary and binary constraints). In



Activation Condition: [A.Age > 5]
Correlation Condition: [T.Sex is female]
Time Condition: 2,5,

Each time A occurs, then B occurs afterwards before A recurs
Positive examples: ABCADB, ACDEB
Negative examples: DDAA, AABCD, EDBAAAC

Figure 15. Editor showing a binary constraint between two activities.

contrast, correlation conditions can only be specified on binary constraints.

All three types of data conditions can be specified by opening the properties panel of the constraint. In this panel, there are sections for activation, correlation and time conditions, as highlighted in Figure 15. Double-clicking on the empty square brackets ’[]’ opens a textbox to specify the condition. After entering the condition, the user can press the “enter” key on the keyboard or just simply click out of the textbox, and this saves the condition automatically if valid. If the condition is not valid, the user gets notified.

To assist the user, we also developed an autocomplete feature that helps to type in the data conditions by providing hints on how to complete the condition based on both the syntactic rules of conditions as well as the information available in the model. For example, for activation and correlation conditions, once modellers type in ’A.’ or ’T.’ (used to refer to attributes of the activation and target activities, respectively), they get different autocomplete options for the attributes available for the corresponding activity, which is then further filtered as the modeller continues typing. An example of this feature, when entering the time condition, can be seen in Figure 15.

5.3.6 Attributes

In the new editor, attributes can be added using the “Add attribute” button in the Control Palette under the “Attributes” section. Clicking on this button opens up a popup “Add attribute” and four fields to fill: Name, Type, Value Range/Possible Values, and Activities.

The popup also has an “Add attribute” button at the bottom to validate and save the currently entered attribute. An example of this popup is shown in Figure 16 with label ’1’.

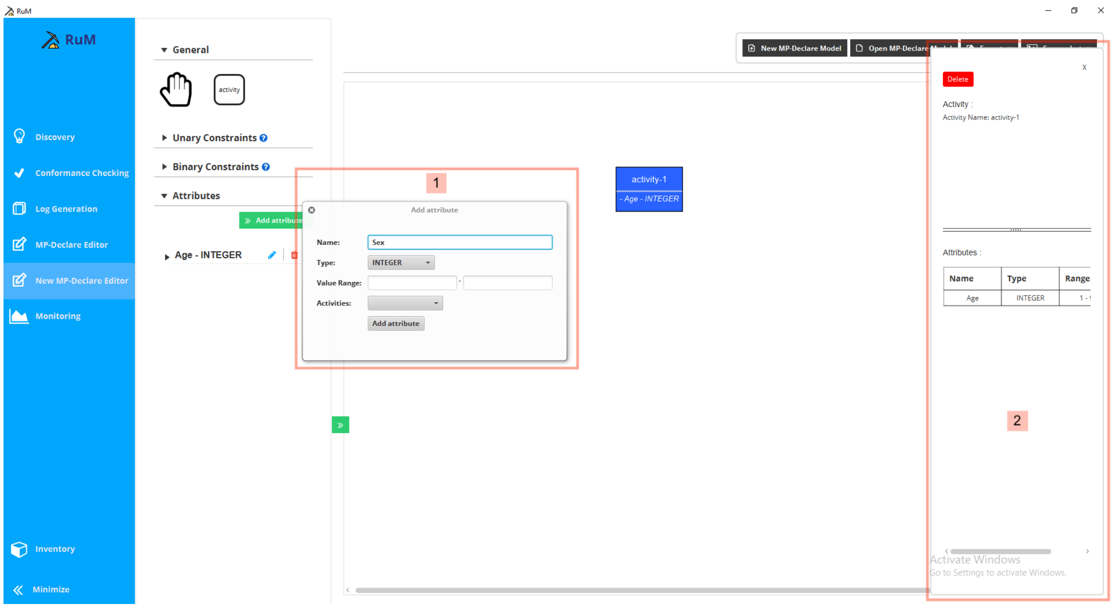


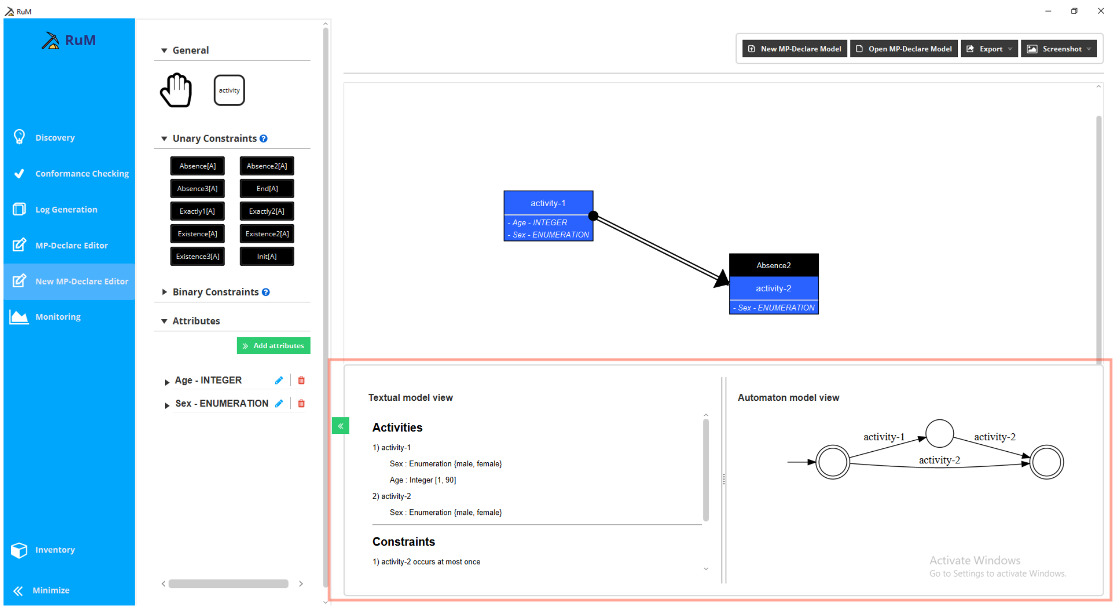
Figure 16. Editor showing attributes.

A saved attribute appears in three parts of the editor. First, in the Control Palette under the attributes section as an entry with an edit button (represented by a blue pencil icon) and a delete button (represented by a red waste bin icon). This entry can also be expanded using the expand arrow on the left of the attribute name to show which activities the given attribute is associated with. Second, in each associated activity in the attribute container on the modelling canvas. Each attribute entry associated with an activity is displayed as a line entry in the attribute container using the attribute name and type. An example attribute entry is $age - INTEGER$, where “age” is the attribute name, and “INTEGER” is the attribute type. And third, the attribute can also be seen in the properties panel of the activity in a table. Figure 16 shows the “Age” attribute displayed in the Control Palette, the activity and the activity properties panel labelled with ’2’ in the figure.

Clicking the edit button on an entry in the command palette opens up a popup “Edit attribute” similar to the creation popup with all the fields prefilled with the existing values. Clicking the delete button removes the attribute and all its associations to activities. An attribute can also be removed from a single activity using the delete button that can be found on each entry in the activity properties panel.

5.3.7 Textual and Automaton Views

In the old version of the Declare editor, there was a single visualisation window that defaulted to showing the Declare model view but could also be toggled to show the Automation view or the Textual model view. In the new Declare editor, all three views can be seen at once. The Declare model can be seen in the canvas, while the Textual and Automaton views can be seen in a bottom panel which can be shown/hidden as necessary. The bottom panel is divided into two halves. The Textual view is shown on the left, while the Automaton view is shown on the right (see Figure 17). These views are updated automatically while the model is being designed.



Textual model view

Activities

1) activity-1
    Sex : Enumeration {male, female}
    Age : Integer [1, 90]
2) activity-2
    Sex : Enumeration {male, female}

Constraints

1) activity-2 occurs at most once

Figure 17. Editor showing textual and automaton views.

Updating the Textual and Automaton view on the fly presented a small challenge. Updating the views on the fly could end up locking the entire editor due to the processing in the background needed to generate the script used to create the graph(in combination with VisJS) from the data in the model. To solve this issue, we created a service to run this processing in a different thread. In this way, we never block the main thread used by the editor, and things can still work smoothly while we run the process to update the Textual and the Automaton view.

5.3.8 Menu Bar

At the top left of the new editor page, there is a menu bar with four buttons: the “New MP-Declare Model” button, the “Open MP-Declare Model” button, the “Export” menu button and the “Screenshot” menu button. The menu bar is shown in Figure 17.

The “New MP-Declare Model” button creates a blank model canvas, clearing the model currently on the screen. Before clearing the canvas, a notification popup appears, warning the user that the current model would be deleted with two buttons to confirm or cancel the action.

The “Open MP-Declare Model” button opens the file directory to select the desired Declare (.decl) file for importing. Once a file is selected, the file is processed, and the model is shown in the Canvas.

The “Export” menu button contains buttons to export the model as a Declare model, as a textual model, as an automaton model or as an XML model (i.e. the model in XML format). The Declare model export produces a .decl file, the textual model export produces a .txt file, the automaton model export produces a .dot graph, and the XML model export produces an XML file (used as input by some older tools for the Declare language). The exported model is saved to the location selected by the user using the name selected by the user.

The “Screenshot” menu button, similar to the “Export” menu button, consists of buttons to take a screenshot of the Declare model, the textual model and the automaton model. Each button takes a screenshot of the corresponding model and saves it with a name selected by the user in a location selected by the user.

5.4 Graphical Information in the Declare File Extension (.decl)

While developing the export and import features, we realised that the .decl file format might need to be extended in order to be used in a graphical editor. The .decl file format was developed in 2018 for MP-Declare by Vasyl Skydanienko in his thesis [24].

One of the drawbacks of the old editor was that the model layout was fully generated by RuM itself, and the user had practically no control over it. Furthermore, the generated layout could be different on every import, even if exactly the same model was used. With the new graphical editor, the user can control the layout of the model, but this also means that any model, when created, exported and then re-imported, should look exactly how the user originally designed it, including the layout that the user specified.

To achieve this, the .decl file format was extended to include layout information about the components in the model. This includes the placement of the activities, the placement of the connection lines, what connector the lines are connected to and the location of any mid-connectors of a connection line. Therefore, the .decl format was extended to include the following keywords:

- connection - represents the connection line and what connector it is connected to on the activation and target activity
- waypoint - represents the mid-connectors on a connection line

- layout - represents the activities and their positions in the canvas.

In order to be backwards compatible and process old .decl files (without layout information), we created an algorithm to generate a layout. For activities, the algorithm creates a graph with the help of Graphviz graph builder using the activity and connections information in the .decl file. The resulting graph tree structure is then combined with estimated activity sizes to layout the activities on the canvas. For Binary constraints, we find the closest available connector on the activation activity and target activity and create the connection line.

6 User Evaluation Results

A qualitative user evaluation study was conducted to evaluate the developed artefact. In this chapter, we will detail the procedure, the user evaluation result and the findings.

6.1 Evaluation Setup

The user study aimed to (1) understand how well users can learn and use the new editor, (2) find problematic or complicated areas in the new editor, and (3) collect suggestions on how the new editor can be further improved.

To achieve this aim, ten process mining experts (all with academic backgrounds and experts in the field of BPM) were selected from various European universities to participate in the user evaluation. Five of the selected participants are Declare experts, while the other five have little to no knowledge of Declare. This split based on knowledge was done to validate that the new editor is both valuable for the experts and, at the same time, easy to learn for users unfamiliar with Declare.

The user evaluation study consisted of the following steps for each participant:

1. An introductory email was sent to the participant to explain what the study was about and the purpose of the study, and to provide a link to schedule a suitable time for the study, a link to the Zoom meeting for the study, and a link to all the files needed (consent form, task list, and input files for the tasks). A redacted example of this introductory email can be found in Appendix I and the consent form in Appendix II

2. A Zoom meeting was started with the participant on the date and time selected by the participant [16].

3. The study was introduced to the participant at the beginning of the test, the consent form was read to the participant, and a verbal consent was asked.

4. The participant was provided with a code and was asked to connect to a remote computer, where a compiled version of RuM was already started [17].

5. The editor interface was briefly introduced to the participant. This introduction was limited in scope to ensure that the participant was not helped or skewed before the study.

6. The participant was asked for permission to record the session.

[16]In every call, there was at least one observer in addition to the facilitator. The facilitator was responsible for guiding the participant through the editor while the observer(s) served a supporting role and intervened in case of more technical questions about process mining or the Declare language.

[17]The access to the remote computer was provided via the remote desktop software AnyDesk[18]

7. The participant was asked to share their screen and start executing the tasks provided earlier in the introductory email. The task list can be found in Appendix III

8. The participant was encouraged to think out loud, ask questions, point out confusing or interesting aspects of the editor, and give improvement suggestions during the test. When the participant got completely stuck, hints were given to help them.

9. When a participant completed the task list, we conducted a short post-interview. The participant was asked about the challenges they faced and ideas or suggestions to improve the editor.

10. The recording of the test was stopped after the post-interview.

11. The participant was sent a link to an anonymous post-survey which measured the participant's experience with the editor during the study. The complete post-survey questionnaire is added to this thesis as an additional file (Appendix IV file "Post-survey.pdf").

The tasks of the user evaluation were divided into two parts. The first part was designed to be an introductory set of tasks for the new editor and focused on building a simple model from scratch. The second part was more complex and involved modifying an already existing and significantly more complex model than the one designed in the first part. For example, the model for the second part contained four times more activities and seven times more binary constraints. This allowed us to evaluate the new editor both in cases where the designed model is almost trivial, and in cases where users need to work with relatively complex models (similar to the ones often resulting from automated process discovery).

6.2 Evaluation Post-Survey Results

The short post-survey was created with the aim of measuring the participant's experience and perception of the new editor. The survey was conducted anonymously to allow participants to answer truthfully with the confidence of not being connected to the answers.

However, this means that the survey results cannot be presented according to the two groups of participants initially identified (BPM experts and Declare experts). Instead, the participants are divided into two groups based on whether or not they have had any previous experience with RuM before the study, as reported by the participants themselves in the survey.

As specified in the method in Section 4.3, the survey was based on the following scales: system usability scale [27], satisfaction [10], expectation fulfilment [10], future use intentions [10] and usefulness [10]. The results of each scale are presented in Table 7.

Table 7. Survey results as averages for all included scales overall and grouped based on previous experience using RuM.

	All participants	Participants with some experience using RuM	Participants with no experience using RuM
System Usability Scale	79.25	83.75	72.5
Satisfaction	4.3	4.5	4
Expectation confirmation	4.05	4.2	3.67
Future use intention	4.22	4.5	3.67
Usefulness	4.69	4.79	4.5

System usability scores below 50 are considered 'Not Acceptable', scores between 51-70 are considered 'Marginal', scores above 71 are considered 'Acceptable' while superior products are expected to score better than 90 points [9].

Given that the average usability score of the new editor (Table 7) is 79.25, we can conclude that the users received the new graphical Declare editor well and the design choices made during the design and development cycles were justified. However, as reported in Table 7, there is a significant difference between the two groups of participants. Participants with some experience using RuM had a significantly higher score, this could signify that, compared to the previous version, the new editor was a lot more pleasing to use. At the same time, this can also indicate that further improvements can be made to enhance the user experience for new users.

The other usability scales (satisfaction, expectation confirmation, future use intentions, usefulness) also have relatively high averages, further confirming the good system usability score. However, it is interesting to note that participants who had previously used RuM were more satisfied than the new users. They regarded the tool to be more useful and would likely continue using it, when compared to participants who did not have any experience with the tool. The expectation confirmation score was also higher for participants with previous experience with RuM, thus indicating that their expectations were met, while the new user experience could be further improved.

6.3 Evaluation Findings

During the study, some problematic aspects of the new editor, and their suggested solutions, were also gathered from the participants. We reviewed the recordings of the tests to gain as much insight as possible from each session with the participants. During the review process, all the details of the user's interaction with the editor that we deemed relevant were written into notes. The notes consisted of the participant's code [19], a short description of the findings, and the suggested solutions, if provided by the participant. The video recordings were about 40 - 60 minutes long on average. The resulting notes were aggregated into an Excel sheet and grouped first by the frequency of participants mentioning a specific issue and later categorised based on functionality and component similarity. This clustering resulted in 18 clusters. This Excel sheet was exported as a PDF file and added to this thesis as an additional file in Appendix IV (file "User-evaluation-report.pdf"). In this section, we will share some of the main findings and feedback from the user evaluation.

Although most of the components of the new editor were received well, many of the participants expected more flexibility when specifying constraint data conditions. For example, participant P3 wanted to be able to add data conditions using a slightly different syntax than the one outlined in Section 3.2.4. This related mostly to the logical operators, for which RuM requires the use of "and" and "or", while some participants were more used to "&&" and "||", respectively. Other participants, for example, P1 and P5, wanted to be able to edit unary constraint data conditions by right-clicking on the constraint, with the expectation of finding an edit button which they could click to start editing the data conditions.

There were also multiple suggestions on how the canvas panning feature should work. At the moment, to pane the canvas, the hand icon has to be active. This was well received by most of the participants. However, some participants commented that this type of user interaction is becoming less common nowadays. Participants P3 and P5 suggested that panning the canvas should be enabled by default without using the hand icon. Participant P6 suggested that the panning should only work if a user clicks on an empty space on the canvas and moves it (otherwise, the clicked element would be moved instead). Participant P9 suggested putting the hand icon on the canvas instead of in the Control Palette, explaining that since this icon controls panning the canvas, it would be easier to see.

There were also some suggestions about the attribute list. Participant P2 suggested that the attribute list should be ordered but was not sure about which ordering to use. This was discussed in more detail with the participant, and several ideas popped up during the conversation. One was to order the attributes using the time they were created, and another was to order the attributes alphabetically. Participant P7 suggested showing just the attribute's name without the list of activities attached to the attribute (currently, activities are shown by default but can be hidden on a per-attribute basis). Participant P10 suggested having a search box to search the attribute list and adding a button to collapse or expand the attribute list.

A very popular suggestion throughout the user evaluation was having the ability to

[19]We created a random code to represent each participant and to preserve the participant's identity, as agreed with the participants in the consent form.

add an attribute to an activity on the activity box itself. Participants P4, P5, P8 and P9, at some point during the second task, tried to add an attribute to an activity by right-clicking the activity box. However, they quickly remembered how they added attributes in the first task (clicking on the "Add attribute" button in the Control Palette). This sparked conversations about what they were trying to achieve by right-clicking in the first attempt. The participants explained that they expected either a context menu or some other UI element to appear, which would allow them to create an attribute for the selected activity. During the discussions, some participants realised that attributes have a one-to-many relationship with activities, which may make implementing this functionality more complicated. In that regard, participant P9 suggested using the existing popup for adding an attribute, and preselecting the activity that the user clicked on to trigger adding an attribute. This would match the expected behaviour while, at the same time, allowing other one-to-many relations to be defined as usual.

Another, somewhat less popular, suggestion was to add more keyboard shortcuts. For example, participants P3 and P4 commented that it would be pretty nice if the modeller could press the delete button or the command + delete (control + delete) button to delete an activity when this activity is selected.

One participant suggested adding the ability to play the automaton view to see how the process could work. Although out of the scope of this thesis, we believe this idea is worth mentioning as it would lead to tighter integration between the different parts of RuM and could serve as a good model explanation tool for new users.

Lastly, an issue that frequently arose was the confusion with precedence binary constraint. We noticed that the constraint type can be confusing because when drawn, the connection line becomes inverted, i.e., the line's direction flips when compared to how all other binary constraints behave. This can be especially confusing for non-Declare experts. We have tried to make it a lot easier to understand in the new editor by adding a preview and an explanation in the constraint properties panel. We also have this explanation specified in the textual model view, and lastly, we have an image on the binary constraint buttons showing the arrow and what direction it would be drawn in. We intentionally added this constraint type to the task list to see how the participants interacted with it, having all these visual aids. As expected, some participants were confused about the direction of the line but quickly used the descriptions to understand if the line was drawn properly or should be drawn again from the other direction. However, it was also clear that there can still be confusion when defining this specific constraint.

7 Conclusion

In this thesis, we presented the new graphical editor in RuM. This is the first visual editor for the MP-Declare extension of the Declare language. With this new editor, users can build Declare and MP-Declare models graphically and in a similar way to modern editors for other process modelling languages. We also presented all the components of the new editor, detailing their functionality and how they look, and provided an overview of the underlying code architecture as well as the technology used for the editor.

Business processing experts (BPM experts and Declare experts) evaluated the new editor, and the result of this evaluation was analysed and presented in this thesis. Suggestions and improvement areas were also identified and presented as part of the thesis. With a usability score of 79.25, which is well above the estimated average of 68 [27], the evaluation result indicated that the tool is usable for novice and expert Declare users. Users with prior RuM application experience were particularly satisfied with the new graphical editor.

For future work, we plan to keep improving the layout algorithm used for Declare models without layout information and then make use of the new visualisation of the editor in other parts of RuM (process discovery and monitoring). We also plan to improve the editor further based on the results of the user evaluation and to submit a paper to the International Journal on Software and Systems Modeling (SoSyM).

References

[1] Wil Aalst, Arya Adriansyah, Ana Medeiros, Franco Arcieri, Thomas Baier, Tobias Blickle, Jagadeesh Chandra Bose R.P., Peter Brand, Ronald Brandtjen, Joos Buijs, Andrea Burattin, Josep Carmona, Malú Castellanos, Jan Claes, Jonathan Cook, Nicola Costantini, Francisco Curbera, Ernesto Damiani, Massimiliano de Leoni, and Moe Wynn. Process mining manifesto. In *Business Process Management Workshops (1)*, volume 99, pages 169–194, 08 2011.

[2] Anti Alman. A desktop application for advanced business rule mining. Masters Thesis: University of Tartu, 2020. Masters Thesis: University of Tartu.

[3] Anti Alman. Lecture: Declarative process mining, April 2022.

[4] Anti Alman, Karl Johannes Balder, Fabrizio Maria Maggi, and Han van der Aa. Declo: A chatbot for user-friendly specification of declarative process models. In *BPM (PhD/Demos)*, volume 2673 of *CEUR Workshop Proceedings*, pages 122–126. CEUR-WS.org, 2020.

[5] Anti Alman, Claudio Di Ciccio, Dominik Haas, Fabrizio Maria Maggi, and Jan Mendling. Rule mining in action: The rum toolkit. In *ICPM Doctoral Consortium / Tools*, volume 2703 of *CEUR Workshop Proceedings*, pages 51–54. CEUR-WS.org, 2020.

[6] Anti Alman, Claudio Di Ciccio, Dominik Haas, Fabrizio Maria Maggi, and Alexander Nolte. Rule mining with rum. In *2020 2nd International Conference on Process Mining (ICPM)*, pages 121–128, 2020.

[7] Anti Alman, Claudio Di Ciccio, Fabrizio Maria Maggi, Marco Montali, and Han van der Aa. Rum: Declarative process mining, distilled. In Artem Polyvyanyy, Moe Thandar Wynn, Amy Van Looy, and Manfred Reichert, editors, *Business Process Management*, pages 23–29, Cham, 2021. Springer International Publishing.

[8] Atlassian. The agile coach. Accessed: 2022-12-17 [Online].

[9] Aaron Bangor, Philip Kortum, and James Miller. Determining what individual sus scores mean: Adding an adjective rating scale. *J. Usability Studies*, 4(3):114–123, may 2009.

[10] Anol Bhattacherjee. Understanding information systems continuance: An expectation-confirmation model. *MIS Quarterly*, 25:351–370, 09 2001.

[11] James Bucanek. *Model-View-Controller Pattern*, pages 353–402. Apress, Berkeley, CA, 2009.

[12] Andrea Burattin, Fabrizio M. Maggi, and Alessandro Sperduti. Conformance checking based on multi-perspective declarative process models. *Expert Syst. Appl.*, 65(C):194–211, dec 2016.

[13] Giuseppe De Giacomo, Marlon Dumas, Fabrizio Maria Maggi, and Marco Montali. Declarative process modeling in bpmn. In Jelena Zdravkovic, Marite Kirikova, and Paul Johannesson, editors, *Advanced Information Systems Engineering*, pages 84–100, Cham, 2015. Springer International Publishing.

[14] Marlon Dumas and Jan Mendling. Business process event logs and visualization. In Sherif Sakr and Albert Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*, pages 398–409. Springer International Publishing, Cham, 2019.

[15] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 854–860. IJCAI/AAAI, 2013.

[16] Gluon. Javafx website. Accessed: 2022-11-14 [Online].

[17] Denizalp Kapisiz. Rule mining with rum. Masters Thesis: University of Tartu, 2019. Masters Thesis: University of Tartu.

[18] Mike Beedle Kent Beck, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001. Accessed: 2022-12-17 [Online].

[19] Marco Montali, Federico Chesani, Paola Mello, and Fabrizio Maria Maggi. Towards data-aware constraints in declare. In *SAC*, pages 1391–1396. ACM, 2013.

[20] Oracle. Introduction to fxml, 2017. Accessed: 2022-10-17 [Online].

[21] Monica Pawlan. What is javafx? Accessed: 2022-11-14 [Online].

[22] Maja Pesic, Helen Schonenberg, and Wil M.P. van der Aalst. Declare: Full support for loosely-structured processes. *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 287–287, 2007.

[23] Rum, rule mining website. Accessed: 2022-10-16 [Online].

[24] Vasyl Skydanienko. Data-aware synthetic log generation for declarative process models. Masters Thesis: University of Tartu, 2018.

[25] Margaret-Anne Storey, Emelie Engström, Martin Höst, Per Runeson, and Elizabeth Bjarnason. Using a visual abstract as a lens for communicating and promoting design science research in software engineering. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '17, page 181–186. IEEE Press, 2017.

[26] D. Tudor and G.A. Walter. Using an agile approach in a large, traditional organization. In *AGILE 2006 (AGILE'06)*, pages 7 pp.–373, 2006.

[27] Usability.gov. System usability scale (sus). Accessed: 2022-12-18.

[28] Wil van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer Berlin, Heidelberg, 2011.

[29] Wil van der Aalst. *Process Mining - Data Science in Action*. Springer Berlin, Heidelberg, 2016.

[30] Vis.js: A simple wrapper for using it on the web. Accessed: 2023-07-11 [Online].

[31] Roel J. Wieringa. Design science methodology for information systems and software engineering. In *Design Science Methodology for Information Systems and Software Engineering*. Springer Berlin, Heidelberg, 2014.

Appendix

I. User evaluation introductory email

Dear ... ,

Thank you for accepting to participate in the RuM user evaluation study. The purpose of the study is to evaluate the usability of a new declarative editor in RuM.

This is a joint study conducted by researchers from the University of Tartu (Estonia) and the Free University of Bozen-Bolzano (Italy).

Below are some details about the test:

- The test will take approximately 45 to 60 minutes, and the test will be recorded.
- Please pick a suitable time for your test using this link - `https://calendly.com/timothy33-tf/60min`. Please write back if there isn't a suitable date for you.
- Your test will be carried out over Zoom via this link - `https://ut-ee.zoom.us/j/92382171427?pwd=a0FYVmNOTEErN05vWUNiK0NrNXVmQT09`.
- A folder containing some files for the test can be found in Google Drive via this link `https://drive.google.com/drive/folders/18LXh9xF11MmwL9-A3e2uCom7HC6xS6bo?usp=sharing`.
    - In the folder, you will find
        - A consent form
        - A list of tasks that will be performed during the test
        - A ".decl" input file for these tasks.
- We will provide you with access to a machine where RuM is installed and the input files are prepared. You will have access via a program called AnyDesk.
- After the test, we will also send you a link to a short survey about your general perception of the editor. The survey will take about 5-10 minutes.

To expedite the test procedure, we kindly ask you to prepare for the test by performing the following tasks (10-15 minutes):

1. Download and install AnyDesk - `https://anydesk.com/en/downloads/`.

2. Print out the task list and have it available for the test.

3. Skim over the task list to get a general idea of the test process, however, I would ask you to not do the tasks before the test.

4. Read, sign, and scan the consent form.

Anydesk is a lightweight solution, so the installation should be fast and easy. You will need a remote address to connect to the test machine. I will provide this at the start of the test.

On behalf of the research team,
Timothy Fadayini

II. User evaluation consent form

Consent to Act as a Participant in a Research Study

Study title: RuM's MP-Declare editor user evaluation

Principal Investigator: Timothy Fadayini

Co-Investigators: Fabrizio Maria Maggi, Anti Alman

Introduction: As an experienced process analyst you were selected to participate in this study. This is a joint study conducted by researchers from the University of Tartu (Estonia) and the Free University of Bozen-Bolzano (Italy).

The aim of this study is to evaluate the usability of the new MP-Declare editor in RuM, a declarative process mining application. We will particularly focus on identifying problematic aspects of the user interface and the general workflow within the editor.

The study focuses on one of the main functionalities of the application which is the MP-Declare editor.

During the study, we will ask you to use the editor to perform process modelling-related tasks. The description of the tasks and the application have been made available to you before the start of your test.

We will record your interaction with RuM and our conversation during the study. After the study, we will ask you a few follow-up questions in a short interview, and we will ask you to fill out a short survey. The results of the test will be used to improve the editor further.

Participation requirements: Any person 18 or older who has experience with process mining.

The expected duration of the study: The study will take about 45-60 minutes of your time.

Risks and Benefits: The risks that are associated with this research are no greater than those ordinarily encountered in daily life. There are no direct benefits to participants, but the researchers anticipate future improvements to the editor to potentially benefit process mining researchers and practitioners.

Privacy and Confidentiality: In order to protect the participants' identities during this study the research team will follow the following procedure: The original recordings will only be accessible to the Principal and Co-Investigators. The video files will be used to analyse the interaction of the participant with the RuM application. The audio contained in the recordings will be transcribed, potential identifiers will be removed or aggregated and the original recordings will be deleted afterwards.
Your data and consent form will be kept separate. Your consent form will be stored securely and will not be disclosed to third parties.

By participating, you understand and agree that the data and information gathered during this study may be used by the participating universities for publication purposes. However, any identifiable information will not be mentioned in any such publication or dissemination of the research data and/or results. The University of Tartu requires all research records to be maintained for at least 5 years following final reporting or publication of a project. Aggregated data will thus be archived by the Principal Investigator for that timespan.

Questions about the Study: If you have any questions, comments, or concerns about the study either before, during, or after participation, please contact the Principal Investigator (`timothy.iyanuoluwa.fadayini@ut.ee`) or the Co-Investigators.

Voluntary Participation: Your participation in this research is voluntary. You may discontinue participation at any time during the research activity. Your decision regarding whether or not to participate in this study will not result in any loss of benefits to which you are otherwise entitled.

I am of age 18 or older. I have read and understood the information above and I want to participate in this study:

☐ Yes ☐ No

Participant: The above information has been explained to me and all of my current questions have been answered. I understand that I am encouraged to ask questions, voice concerns or complaints about any aspect of this study during its course and that such future questions, concerns or complaints will be answered by a qualified individual or by the investigator(s) listed on the first page of this consent document.

Co-investigator: I certify that I have explained the nature and purpose of this research study to the participant and discussed the potential benefits and risks of study participation. Any questions the participant had about this study have been answered, and we will always be available to address future questions, concerns or complaints as they arise. I further certify that no research component of this study has started before this consent form was signed.

Participant	Co-investigator
Date	Date

‘

III. User evaluation task list

Introduction
You work in a hospital and the manager of the hospital has just been made aware of a new process modelling language that is constraint-based which could simplify the representation of the hospital's processes. To test this out, you have been given access to an MP-Declare editor in a tool called RuM. Using the sepsis case as a background you have been given a set of tasks to test out this editor.

Task 1
We will begin by drawing a simple model. Introducing the various components available in the model editor.

1. Add a new activity and rename the activity - **"ER Registration"**
2. Add another activity and rename the activity - **"ER Sepsis Triage"**
3. Add another activity and rename the activity - **"Admission NC"**
4. Add a new attribute
   (a) Name: **Age**
   (b) Type: **Integer**
   (c) Range: **20 - 90**
   (d) Activities: **ER Registration, ER Sepsis Triage**
5. Add another attribute
   (a) Name: **Diagnose**
   (b) Type: **Enumeration**
   (c) Values: **DD, YC, UC, MA, MD**
   (d) Activities: **ER Registration, ER Sepsis Triage, Admission NC**
6. Add an **"Alternate Response"** binary constraint between **ER Registration** and **Admission NC**, such that ER Registration is followed by Admission NC.
7. Add a **"Succession"** binary constraint between **ER Registration** and **ER Sepsis Triage**, such that ER Sepsis Triage follows ER registration.
8. Add a **"Precedence"** binary constraint between **ER Sepsis Triage** and **Admission NC**, such that ER Sepsis Triage occurs if preceded by Admission NC.

9. Add Unary constraint **“Existence[A]”** to the **ER Sepsis Triage**
    (a) Activation condition: **A.Age > 24 and A.Age < 70**
    (b) Time condition: **2,5,h**
10. Add Unary constraint **“Init[A]”** to the **ER Registration**

Task 2
Now let’s make some modifications to a more complex model using the editor.

1. Import/Open the Sepsis model provided to you beforehand.
2. Remove the **Alternate Response** constraint between **ER Sepsis Triage** and **CRP**.
3. Add the following attribute to activity ER Triage
    (a) Name: **Treatment**
    (b) Type: **Enumeration**
    (c) Values: **T1, T2**
4. Change constraint **Alternate Succession** [ER Triage, ER Sepsis Triage]
    (a) Template to: **Response**
    (b) Activation condition: **A.Treatment is T1**
5. Export Declare model.

IV. Additional files

1. Post-survey.pdf
2. User-evaluation-report.pdf

V. Licence

Non-exclusive licence to reproduce thesis and make thesis public