

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

Iryna Halenok

Business Process Simulation with Differentiated Resources

Master's Thesis (30 ECTS)

Supervisors: Orlenys López-Pintado, PhD
Marlon Dumas, Professor

Tartu 2023

Business Process Simulation with Differentiated Resources

Abstract:

Business process simulation is an approach that allows us to perform the "what-if" analysis. With its help, we can analyse the current business process, manually find possible improvements, introduce them, and predict the impact of those changes by running a simulation. Simulation tools take business process models as input, accompanied by the additional details required for simulation, such as resource availability. Yet existing simulation tools support only basic process elements, among which are activities and decision points (gateways). In real life, however, a resource can perform activities not straight after enabling time but instead waiting for a group of activities to gather and then execute them in one go (batch processing), or a resource can prioritise one task over another when both of them are waiting for the execution (task prioritisation). Additionally, process simulation might benefit from introducing events to model various behaviour, for example, setting up a timer for 2 hours or interacting with external entities like calling a client or receiving a message from a client. We call these types of events intermediate events as they happen during a process. This thesis contributes to implementing those concepts above (batch processing, task prioritisation, intermediate events) based on the already implemented simulation engine with differentiated resources. Furthermore, the simulation engine we use as a basis, named Prosimos, does not have a web interface and can only be executed through the command line interface. This, in turn, has limited the adoption of Prosimos in practice. With this thesis, we also aim to diminish the knowledge requirement and allow people with no technical background, like business analysts, to utilise the tool. To achieve this, we implement a brand-new web application from ground up. During the development process, we write unit and integration tests, following the decision table testing approach, to verify the implementation continuously. For evaluation, we analyse the scalability of the newly introduced concepts. The results of this master's thesis were already partially published as a demo paper.

Keywords: Business Process Simulation, Batch Processing, Task Prioritisation, Intermediate Events, Web Application

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

Äriprotsesside simulatsioon diferentseeritud ressurssidega

Lühikokkuvõte: Äriprotsesside simulatsioon on lähenemine, mis võimaldab meil teha "mis-oleks-kui" analüüsi. Selle abil saame analüüsida praegust äriprotsessi, käsitsi leida parendusvõimalusi, neid modelleerida ja seejärel ennustada nende mõju simulatsiooni abil. Simulatsioonitööriista sisendiks on äriprotsessi mudel, millele on lisatud simulatsiooniks vajalikud lisaandmed, nagu näiteks ressursside kättesaadavus. Samas toetavad olemasolevad simulatsioonitööriistad vaid protsessi põhielemente, nagu näiteks tegevused ja otsustuspunktid (löösid). Reaalses elus ei pruugi ressurss aga tegevusi teha mitte kohe pärast seda kui tegevused muutuvad lubatuks, vaid selle asemel oodata, kuni koguneb suurem grupp ootel tegevusi, mille saab seejärel ühe korraga teha (partii töötlemine). Või kui mitu tegevust on samaaegselt ootel, siis võib ressurss seada ühe tegevuse teise ees esikohale (ülesannete prioritseerimine). Lisaks võib äriprotsesside simulatsioon kasu saada täiendavate käitumiste modelleerimiseks vajalike sündmuste toetamisest. Näiteks taimeri seadistamine 2 tunniks või väliste osapooltega suhtluse modelleerimine (kliendile helistamine, kliendilt sõnumi vastuvõtmine jms). Me nimetame seda tüüpi sündmusi vahepealseteks sündmusteks, kuna need toimuvad protsessi täitmise käigus. Käesolev töö panustab eespool nimetatud kontseptsioone (partii töötlemine, ülesannete prioritseerimine, vahepealsed sündmused) implementeerimisse kasutades juba olemasolevat simulatsioonimootorit, mis toetab diferentseeritud ressursside seadistamist. Samas ei ole aluseks võetud simulatsioonimootoril, nimega Prosimos, veebiliidest ja seda saab kasutada ainult käsurealiidese kaudu. See on aga omakorda vähendanud Prosimose kasutuselevõttu praktikas. Käesoleva lõputöö eesmärk on lisaks vähendada käsurealiidese kasutusest tulenevat tehniliste teadmiste eeldust, et seeläbi võimaldada tehnilise taustata inimestel, näiteks ärianalüütikutel, antud tööriista kasutada. Antud eesmärgi saavutamiseks arendame nullist täiesti uue veebirakenduse. Veebirakenduse järjepidevaks arenduse käigus kontrollimiseks kirjutame üksuse- ja integratsiooniteste, lähtudes otsustustabeli testimise lähenemist. Loodud tarkvara hindamiseks analüüsime kasutusele võetud kontseptsioonide implementatsiooni skaleeruvust. Käesoleva magistritöö tulemused on osaliselt juba demo artiklina avaldatud.

Võtmesõnad: Äriprotsessi Simulatsioon, Partii Töötlemine, Ülesannete Prioritseerimine, Vahepealsed Sündmused, Veebirakendus

CERCS: P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Contents

1	Introduction	5
2	Background and Related Work	9
2.1	Business Process Management	9
2.2	Business Process Management and Notation	9
2.3	Business Process Simulation	11
2.4	Batch Processing	12
2.5	Prioritisation	14
2.6	Existing Simulation Solutions	14
3	Software System Description	16
3.1	Web Application Architecture	16
3.2	System Architecture	19
3.3	CI & CD pipelines	23
4	Simulation Enhancements	26
4.1	Intermediate Events	26
4.2	Batch Processing	29
4.3	Case-based Prioritisation	42
5	Testing	53
5.1	Case-based Prioritisation	53
5.2	Batch Processing	55
5.3	Code Coverage	57
6	Evaluation	59
6.1	Datasets	60
6.2	Experimental Setup	60
6.3	Results	62
7	Conclusion and Future Work	66
	References	68
	Appendix	69
	I. Licence	69

1 Introduction

Business process simulation (BPS) is a technique for performing a "what-if" analysis. This type of analysis helps with knowing what will happen in a real-life under changing some parameters so that one prepares to face similar situations in the future. The formation of the question is the first step needed, for example, "What will be the impact on the process cycle time if one additional working hour for the resource X on day Y is introduced?".

The simulation requires a BPS model as an input. Accordingly, it contains a business process model and a description of the simulation scenario. There are different perspectives we can use while modelling a business process. One of them is the control-flow perspective, which specifies the order and execution condition of activities and events [DRMR18]. In Business Process Model and Notation (BPMN), an activity is an item of work performed by a resource. An event depicts an instant occasion which does not encompass any duration. Examples of events are receiving a loan application from a client, sending a booking confirmation to a client or waiting till the fifth of every month. We connect events and activities with the help of sequence flows to show the execution order. Additionally, we use gateways, representing decisions and alternative business process paths. They help illustrate situations when we need to perform two tasks in parallel (AND gateway) or one of two possible tasks based on some condition (XOR gateway).

The second component of a BPS model is a simulation scenario description. Its details depend on each separate implementation. In general, it includes the following sections: 1) scenario specification (how many process instances to generate and start date and time of a process), 2) arrival calendar (time intervals during which cases arrive), 3) arrival rate (distribution describing how often new process cases arrive), 4) resource calendars (time intervals during which a given resource is available to perform tasks), 5) resource profiles (description of number, cost and working schedule of resources), 6) resource allocation (distribution describing how a given resource perform assigned tasks), 7) branching probabilities (probability of every possible path taken in case of a gateway).

The simulation results can be presented differently: either the simulated event log file, the list of multiple key performance indicators (KPIs) and their values for the business process, or some different output. In the current work, we will be mainly concerned with the results containing both the simulated log file and multiple key performance indicators (KPIs). The performance metrics represent the whole process in general and individual activities and include *waiting time* – the duration from the moment activity is enabled until it is started; *processing time* – the duration between the beginning and end of an activity instance; *cycle time* – the difference between the end time and start time of a process case; and *resource utilization* – the ratio of the available time of a resource spent executing process activities [LPHD23]. As a rule, the list of metrics varies from

simulator to simulator; we present here the common metrics all simulators try to support.

The description of the simulation scenario impacts to what extent the simulation results are realistic. The better the scenario is described at this point, the better simulation results we will receive. For example, the simulation scenario contains four main sections specified earlier and does not include a section for describing intermediate events. This results in the tool's inability to simulate intermediate events when the real business process contains it, or analysts want to introduce it to a business process. As analysts make decisions based on the received output from simulation, we aim to ensure that the simulation engine reflects real-world scenarios to the best of its ability.

Mainstream simulation tools support basic BPMN elements, among which are activities, events and gateways. One of the open-source simulation engines, namely *Prosimos*, working with BPMN, presented in [LPD22], provides support for resources who perform every task at their own speed (differentiated performance) and have their own working schedule (differentiated availability). This concept is introduced under the name of *differentiated resources*. On the contrary, *undifferentiated resources* are those who share the same performance and availability inside resource pools. In [LPD22], the novel approach is evaluated and recognised as the one that outperforms the undifferentiated resources. Due to this advantage, we use *Prosimos* as a baseline for the thesis work.

However, previously mentioned solutions still lack the support of some real-world use cases. Assume that 1) the resource is not going to execute the task when it became enabled, but one waits and performs the work after some time (e.g., the clerk waits till 5 pm in order to document all invoices performed during the day) or 2) one task is being prioritised over another one under a condition/rule (e.g., the client inquiries with status "Critical" has the highest priority and starts to be processed as fast as possible). Neither batch processing (first assumption) nor task prioritisation (second assumption) has support in those simulation engines. Yet both of those concepts happen in real life. Hence, this defines one of the limitations of the simulation engines mentioned above.

Moreover, *Prosimos* does not support intermediate events yet. We use events to describe situations when something happens instantly, or, for example, we communicate with entities from inside or outside of an organisation. End-users may find it inconvenient since events are commonly used during the modelling of a business process. For example, [zMR08] discovered that one of the subsets of intermediate events, namely *intermediate messages*, are used in 41% of analysed models, used for education purposes, and in 12% of consulting models. Adding support for intermediate events allows us also to introduce support for event-based gateways. Those gateways are used when we need to react based on which event happened faster. By enabling the usage of intermediate events and event-based gateways, we allow users to simulate and analyse more use cases. Due to those reasons, we aim to address the absence of intermediate events' support as a part of this thesis.

Besides impediments in a control flow (no support for intermediate events) and a

BPS model (no support for batch processing and task prioritisation), [LPD22] also lacks the user interface (UI) for a better user experience. The existing solution is presented as a command line interface (CLI) tool. This introduces several obstacles for an end-user: 1) not being able to use the tool in case of no technical knowledge regarding running Python scripts; 2) introducing changes requires one to modify a raw JSON file; 3) complexity of some sections of a simulation scenario, which requires one to specify a unique identifier of BPMN elements.

In this thesis, we fill the mentioned gaps and focus on the following research goals:

RG1: *Extend the simulation tool by introducing intermediate events, batch processing and case-based prioritisation*

RG2: *Develop and deploy a web application for simulation, allowing users to specify a BPS model and view simulation results*

In order to accomplish these goals, we utilise a design science research methodology. [HMPR04] states that using this methodology allows building an artifact and performing its evaluation. As our goal is to solve a problem by presenting a new iteration of *Prosimos* tool, we find this methodology a great fit for our purpose.

Within the design science methodology, we adopt phases of the systems development life cycle (SDLC). In general, SDLC was introduced to provide a structured view of system development in order to meet clients' requirements. The list of phases includes the following ones: requirements gathering and analysis, designing, developing, testing, deploying and maintaining software. ISO/IEC/IEEE 12207 standard¹ refers to SDLC as a framework. This implies that SDLC does not force a development team to perform any specific activities. The main goal of SDLC is to document a possible and nice-to-have list of activities which makes the development efficient. A development team is responsible for selecting which of those activities suit the team and incorporating them into the development process (with changes if needed). In this thesis, we follow the same approach and choose all phases because they match our goals. Additionally, SDLC does not require those stages to be executed in a particular order.

During our work, the development is conducted following Agile practices. We plan to adopt one of the most popular Agile frameworks - Scrum [SS20], which is an iterative and incremental approach. First, we define initial requirements before starting the coding itself. Afterwards, we proceed with the implementation based on the requirements, and then we re-iterate the requirement specification and implementation. This approach allows us to develop the features iteratively and change the specifications if needed.

The version control system used during the development is Git with GitHub² as a hosting service since *Prosimos* [LPD22] uses it. We also leverage GitHub functionalities

¹<https://www.iso.org/standard/63712.html>

²<https://github.com/>

and use GitHub for requirements documentation and task planning. Additionally, we configure pipelines (GitHub Actions³) for continuous integration (CI) and continuous delivery (CD). CI executes the build and performs testing, whereas CD releases the code, allowing us to deploy a new version at any moment we want.

Furthermore, we use *Grammarly*⁴, an AI writing assistant, during the writing process in order to verify the grammar and punctuation and increase the understandability of the written work.

The rest of the thesis is structured the following way. Section 2 gives an overview of this work's fundamental terms and concepts. Additionally, in this section, we describe what has been done in the field so far. Section 3 introduces an overall architecture from two points of view: as a web application and a functional system. Furthermore, we explain the CI/CD pipeline, which contributes to the maintenance easiness later. Section 4 describes requirements, design and implementation of newly introduced concepts to *Prosimos*, namely intermediate events, batch processing and case-based prioritisation. Section 5 provides details on how we test the developed artifact. In section 6, we examine how the new iteration of the software scales up. Section 7 summarises what has been done in the thesis's scope and discusses possible improvement areas.

³<https://github.com/features/actions>

⁴<https://www.grammarly.com/>

2 Background and Related Work

This section provides an overview of concepts that are relevant to the area of research in the scope of this thesis.

2.1 Business Process Management

According to [DRMR18], Business Process Management (BPM) encompasses principles, methods, techniques, and tools in order to maintain the business process life cycle. The life cycle includes different stages of a business process: identifying, discovering, analysing, redesigning, implementing, executing, monitoring, and adapting. BPS is a technique which belongs to the analysing stage. Simulation allows us to run a business process and acquire numerical data, particularly performance metrics (for example, cycle or processing time). Consequently, simulation represents a quantitative process analysis.

Additionally, we use the discovery of input parameters in the simulation engine. However, this work does not implement this part; we use an already existing solution for this described in [LMCCD22]. The discovery tool uses a business process model and an event log file as input. *Event log file* describes the executed cases where case refers to one execution of the business process. As the name suggests, this log file lists events that happened during the whole process execution. Each event is described with the help of attributes. The list of attributes is not fixed and depends on an automation system which is used by the business. In this thesis, we focus on the following attributes: case identifier (unique identifier of the case), event name (refers to the name of executed task or event), resource (name of the resource who executed the task), enabled time (timestamp when the task was ready to be executed), start time (timestamp when the task was started being executed by the resource), end time (timestamp when the task was finished).

2.2 Business Process Management and Notation

One of the essential parts of BPM is depicting a business process so that all stakeholders (people involved) understand it. For this, we use *process models*. There are various modelling languages used for presenting a business process. In this work, we stick to the de-facto standard of business model diagrams - BPMN [Gro13], which is widely used and supported.

BPMN is a graphical notation used to visualise a business process. This notation contains various elements to denote a business process. Key elements of this notation include the following items:

- *activity* is a work item executed by a resource. A resource is anything, be it a human or a machine, which can execute the task [DRMR18]. In this work, we use *task* and *activity* words interchangeably.

- *event* is something that happens instantaneously. For example, a bank receives a loan application form, or a client receives a loan rejection letter. In BPMN, there are three types of events: start, end and intermediate events. Start and end events happen at the beginning or end of a business process accordingly. A start event signals the creation of the process instance, while an end event indicates the finish. Concerning our examples, receiving a form might be a start event, and receiving a rejection letter might be one of many possible variations of a process end. At the same time, both of those examples might represent an intermediate event. An intermediate event happens *during* a business process. Additionally, all events are organised into two types: catching, being able to catch a trigger and throwing, being able to throw a trigger. Here, *trigger* specifies a nature (origin) of an event, for example, message, error, escalation and others.
- *gateway* is a decision point that chooses which path to follow. Those decisions are either data- or event-driven. For example, we might follow a different path in case a loan application amount is higher than 10 000 euros (data-driven), or we need to send a payment reminder to an end-user if payment is overdue (event-driven).
- *arc, or sequence flow*, allows us to specify the order of those items we mentioned above. Usually, an activity or event, or gateway usually has an incoming and outgoing sequence flow. An element might have one or multiple incoming or outgoing flows depending on the element type.

In order to have a better overview of process instance execution, we introduce the term of *token*. Tokens are used to control the current state of each process instance running [DRMR18]. As a result, tokens move during execution. When a start event is triggered, we instantiate a process case and place a token on the event outgoing arc. This token enables the next element located down the sequence flow. Let us suppose that the next element is an activity. We call activity *enabled* when one or many token(s) are placed on its incoming flow(s) [LPD22]. An activity is not started till the moment there is an available resource to execute it. Once we have a resource, we start processing an activity and a token is placed inside an activity. After a resource finishes executing this activity, the token is placed on the activity's outgoing sequence flow(s). Once there are no tokens in any arcs, the process instance is finished. In regard to this so-called token game, events could not contain tokens inside themselves during execution, compared to activities. This happens due to the fact that events, basically, do not have any duration; they are executed instantly.

Apart from having a gateway as a key element, BPMN also defines multiple types of gateways, including the following:

- exclusive (XOR) gateway represents alternative paths of a process. During the execution, only one outgoing sequence flow must be chosen and followed.

- parallel (AND) gateway allows describing situations when a process executes some tasks in parallel (concurrently) to each other. In this case, all gateway's outgoing sequence flows are being followed. Decision of both AND gateway and XOR gateway depends on the process data. Commonly, gateways are named with a question, and each of the outgoing arcs is an answer to a posed question. So, the path is selected based on the answer to a question.
- event-based gateway depicts cases when we choose which path to follow based on the earliest event that happens. For example, we use this gateway when our response differs based on a type of intermediate event. There are some limitations on which elements you can use after an event-based gateway. Allowed elements include intermediate catching events or receiving activities. We also might refer to those following events as *external catch events* due to the fact that they usually happen outside of a business process.

In this work, we focus on types of intermediate events depicted in Figure 1a. *Message* intermediate catch event has a message as a trigger, for example, an invoice from a bank. *Timer* represents a case when we need to wait based on an absolute or relative time; for example, every Monday (absolute) or 24 hours (relative). *Signal* event is used together with a publish-subscribe service [DRMR18]. This element waits to be triggered by some other process or service. Additionally, we implement an *event-based gateway*, depicted in 1b. We provided its definition earlier in this section.

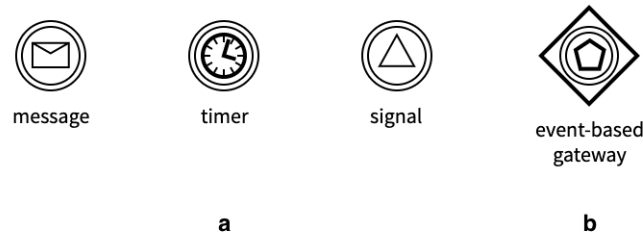


Figure 1. Supported **a)** intermediate catch events and **b)** event-based gateway in *Prosimos 2.0*

2.3 Business Process Simulation

Improving a business process running in the real world might be a challenging, costly, and time-consuming task. The simulation is a technique that allows finding the best setup of the improvements instead of hand-picking and implementing them in real life. This approach allows us to instantiate the desired number of cases with the specified simulation

scenario, execute those process cases, receive quantitative results of a simulation, analyse them and then change the simulation scenario, run the simulation again and compare whether the results improved.

The simulation receives a business process model and a simulation scenario as input. Since our baseline is *Prosimos* [LPD22], the process model is specified by the usage of the BPMN modelling language. Another parameter is the simulation scenario which includes the next main sections:

1. *Scenario Specification* includes 1) the number of process cases to be simulated and 2) the start date and time of the simulation.
2. *Arrival Calendar* lists time intervals during which process cases are allowed to arrive. An individual *time interval* contains four attributes: start weekday and time and end weekday and time. For example, from Monday, 9 am to Monday, 9 pm is a time interval.
3. *Arrival Rate* describes how often process cases arrive. We describe this with the usage of distribution functions. For example, cases arrive following a normal distribution with a mean of one hour and a standard deviation of 10 minutes.
4. *Resource Calendars* defines a list of time intervals during which a given resource is allowed to perform process tasks. The definition for a *time interval* is the same as used for *Arrival Time* section.
5. *Resource Profiles* represents all available resources grouped into pools. Each resource has the following attributes: identifier, resource name, cost per hour, and resource amount.
6. *Resource Allocation* maps available tasks to resources that are able to execute this task. One task may be assigned to one or multiple resource profiles. In the case of several allocations for the same task, each resource has its own individual performance. We describe resource performance with the help of distribution functions.
7. *Gateway Branching Probabilities* defines probabilities for all outgoing flows of a gateway. Every gateway has one or multiple outgoing flows. The allowed range of probability is from 0 to 1. The sum of all probabilities of a gateway should be equal to 1.

2.4 Batch Processing

A resource might behave differently when selecting which task to execute next. Sometimes, a resource does not execute an enabled task straight away. Instead, a user or

software waits for a group of items to be collected and only after that runs them together. We call this *batch processing*. [RATE05] defines multiple resource patterns of the workflow systems in general. One of them is *Piled Execution* which directly refers to batch processing. The pattern describes allocating and queuing multiple instances of the same task from different process cases to the same resource. Consequently, we introduce an additional condition, besides resource availability, for an activity to be assigned to a resource and executed. In this paper, we call those additional conditions - *firing rule* or *batch activation rule*. In case an activity is batched, we collect those activities in the queue during the execution of the business process till the moment the firing rule is true. Afterwards, if 1) the firing rule is true and 2) the resource to execute a batch is available, we assign a resource to the task and start execution. The introduction of this concept helps to optimise task execution by impacting processing time [RM05] and/or cost [RM05, PW13]. For example, if the individual task requires a long time of preparation (e.g., heating up the machine) before the actual execution, piling up multiple tasks saves us some time during the process execution.

In our case, we use the description of the already identified batches from the real-life logs as a part of the simulation scenario. The batch processing description is structured in the following way:

1. *Task Identification* specifies a unique identifier of an activity. This identifier should match the one specified in a BPMN file of a model.
2. *Batch Type* describes how activities are executed in regard to each other inside a batch. [LMCCD22] described five batch processing types: parallel, sequential (task-based and case-based) and concurrent (task-based and case-based). Similarly, [MS06] identified parallel and sequential types of batching. In this paper, we work only with task-based batching and choose parallel and sequential types as an overlap of those two referenced works. [MS06] describes those types as follows:
 - sequential batching requires the initial set-up before the task execution, e.g. warming up a machine. After this, all tasks are executed one after another and the next task could not be started before the previous one ends.
 - parallel batching happens when we can execute our tasks simultaneously and we have no detailed information on how the resource/system split the efforts between all of them. For example, this might mean that our machine handles processes in a multi-threaded way and executes a batch of activities at the same time.
3. *Duration Distribution* specifies the activity duration in the batch processing.
4. *Firing Rule* describes the conditions when batch processing is enabled for the execution. There exist a lot of types of batch activation rules. [HB94] defines

time, *quantity* and *time-quantity* rules. In other words, [LMCCD22] specifies them as volume-based (batching is based on a number of accumulated tasks) and time-based (batching is based on an absolute time or a waiting time limit). One of our intention during this work is to allow users to incorporate the automated BPS discovery tool [LMCCD22] together with *Prosimos*. Due to this, we support those rules which can be discovered using [LMCCD22]. So the proposed solution supports volume-based (*size*) and time-based (*waiting time since the first activity*, *waiting time since the last activity*, *day of the week*, *hour of the day*) rules.

2.5 Prioritisation

In order to perform a task, we need, first, to have a resource that can perform this task. Resources, in turn, may apply some logic when deciding which items to start working on. While explanations of resource behaviour concerning task selection are not covered much in the literature, resource decisions impact the overall performance of a business process [HLD12]. Prioritisation describes resource behaviour when one assigns a higher priority to an activity or case based on rules or logic presented in any other form. [SWX⁺17] approached this topic from a more descriptive perspective by analysing how the resources prioritise their work.

As a rule, queues are being used in simulators to maintain the task selection process. However, there are different types of queues which can be incorporated. One of the possible queues is a priority queue, deciding the next item to execute based on a priority key. In simulators, enabled time is a priority key. Changing the resource selection behaviour implies introducing another variation of a queue in a simulator.

2.6 Existing Simulation Solutions

While many concepts we mentioned before are still under research, some simulation tools already work with either batching or prioritisation. Alongside, intermediate events are supported by all simulators we mention in this section, namely *iGrafx*, *BPSim*, *BIMP*, *Bizagi*, *BonitaSoft*, *Visual Paradigm*, *ARIS Business Simulator*.

Based on [DBBMB16] analysis, only eight out of 33 evaluated simulation tools permits batches. The low support number highlights that batch processing needs more coverage in the simulation tools. Furthermore, all eight tools are proprietary; consequently, code sources are not shared with the public. We pick one of the tools, namely *iGrafx*, and analyse how batching is supported there. *iGrafx* is selected as it is the only tool that both enables batching and is highly evaluated in all segments based on ranking features presented in [DBBMB16]. The list of options by which a batch can be formed is extensive and includes size (number of items in a batch), expression, time, message,

signal, and case attribute value⁵. *iGrafx* supports maximum waiting time for collecting items, which we call waiting time since the *first* item in our work. However, the concept we introduce, calling waiting time since the *last* item, is not possible to design in *iGrafx*. We describe definitions of waiting time since the first/last item later on.

[FP15] singles out *BPSim* simulation tool as the only one which can handle priorities in comparison to the inability of *BIMP*, *Bizagi*, *BonitaSoft*, *Visual Paradigm*. *BPSim* allows adding priorities per task only; a user assigns a priority number to a task with respect to other tasks. Since the publication of [FP15], *Bizagi* and *BonitaSoft* introduced priority, following the same approach as in *BPSim*. Those tools are oriented on prioritising specific tasks, while we want to prioritise all tasks of a specific case instance. *ARIS Business Simulator* allows users to prioritise cases by assigning a priority to the start event. The priority of the start event will be propagated to all activities of this process instance⁶. Similarly, *iGrafx* enables users to prioritise cases (which they call transactions). However, we want to support prioritising cases based on the case attributes which are being generated during the simulation. Thereby, end-users do not provide any priority numbers; software calculates this priority based on rules from the users.

⁵The list was taken from the *iGrafx* documentation

⁶https://documentation.softwareag.com/aris/Designer/10-0sr6/yad10-0sr6e/10-0sr6_Method_Manual.pdf

3 Software System Description

This section describes one of the two main contributions delivered in the scope of this thesis: the creation of the web application (**RG2**). First, section 3.1 explains the overall architecture of the web-based simulation tool Prosimos. The next section presents the system architecture in regard to the implemented functionality. We compare functional components before and after implemented changes in this thesis. Section 3.3 describes the process of verifying and releasing a new version of the web application. We perform these actions as part of the tool maintenance.

3.1 Web Application Architecture

Prosimos 2.0 is a web-based tool developed as a part of this thesis. In this paper, we refer to a version of the tool available before the start of this thesis as *Prosimos 1.0* and a version implemented as a part of this thesis - *Prosimos 2.0*. Besides developing the tool itself, we also deploy it. The deployed version of the tool is available at <https://prosimos.cloud.ut.ee/>. Consequently, we design the architecture of the web application, taking into account the deployment need. Due to the popularity of the approach and its scalability, we dockerize the web application to ease its deployment. Figure 2 depicts the architecture design of the application, which implicitly includes three levels: front-end (client side), back-end (server side) and data layer (data storage). However, explicitly, we have five Docker⁷ instances. In our case, *Web API*, *Broker* and *Task Management* containers represent the server side.

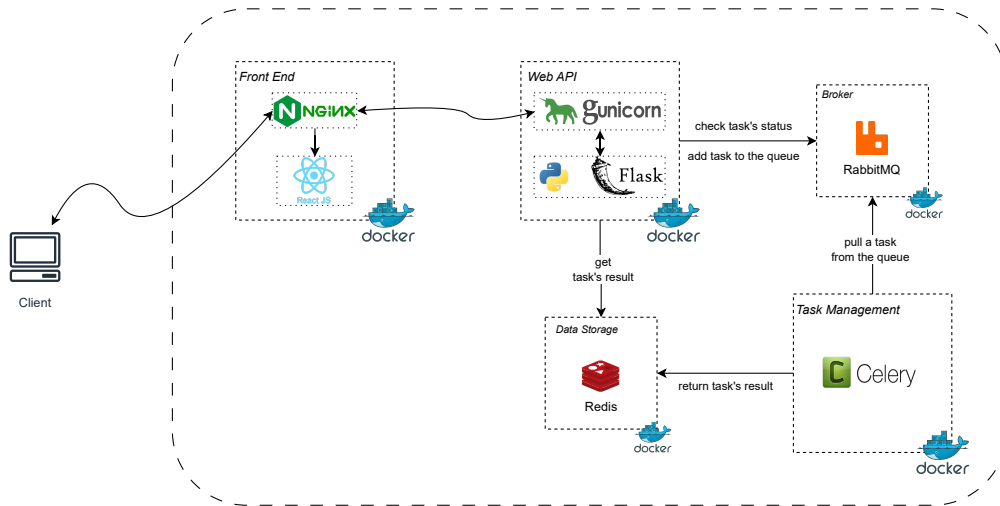


Figure 2. Web application architecture of Prosimos 2.0

⁷<https://www.docker.com/>

Apart from depicting the containers, Figure 2 also illustrates the technology stack we use. The CLI tool *Prosimos 1.0*, the basis for this work, is implemented in Python. Therefore, we continue building the tool for our back-end using Python. One of the thesis tasks is exposing our CLI tool to the public as a web server. For this, we select Flask framework, which, according to [Sem23], has the following advantages: easiness of learning, fitting small projects and high scalability. *Prosimos* client is developed with the help of React library. We choose it based on the advantages presented in [Mur23]: components' reusability, ease of learning compared to Angular, an active community, and fast performance.

The requests flow is happening the following way. First, a client sends a request. It is being received by *nginx*⁸, which performs the role of a reverse proxy. The proxy resolves the request by sending it either to the *React* application, serving static files, or to the back-end (*Flask* application), serving REST API. Additionally, reverse proxy secures our REST API endpoint from being exposed to the public. Once we serve the page to an end user, they can interact with the page and its functionality. Some of the functionality (for example, simulation run) requires calling *Web API* to process a request. This time, *nginx* redirects the request to the server side. On the server side, the first access point is *gunicorn*⁹, it transforms the request received from a web application and sends it to the web server, *Flask* application. After request processing, *Flask* returns the answer the same way as the request sent to the server: first, to *gunicorn*, followed by *nginx* and finally to the client.

Since we are building a web-based tool, one of our goals is to allow multiple users to use the tool simultaneously. As a result, our Web API component should work asynchronously, meaning not blocking access for the next user while processing the request for the previous one. Celery¹⁰ task queue manager is being introduced in order to fulfil this requirement. Figure 2 presents a detailed view of how tasks are being handled in the system. Once the task reaches the Flask web application, it is sent to the message broker (queue). A message broker is responsible for collecting all incoming requests, placing them in the queue and saving the status of the request (e.g., whether the task waits for the execution, is being executed now or (un)successfully finished). We use Rabbit MQ¹¹ as a message broker due to its ability to guarantee delivery of the message and deliver larger files, compared to alternatives, such as Redis. *Task Management* instance runs a set of celery workers. Those celery workers are a set of processes which are responsible for executing tasks independently. Once there is a task in the queue, a worker will pull it out from the queue and start the execution separately from the main application process. This results in a non-blocking execution of long-running tasks. When the task finishes, the results are sent to the Results Backend in case it is set up.

⁸<https://www.nginx.com/>

⁹<https://gunicorn.org/>

¹⁰<https://docs.celeryq.dev/en/stable/>

¹¹<https://www.rabbitmq.com/>

There might be situations when one does not need to save the results of the execution and this step will be omitted. Our use case implies the saving of the results because process simulation results in files with simulation statistics and simulation logs. We use Redis¹² for saving the results of the task execution. Redis is an in-memory key-value data storage which provides an efficient way to save messages of small sizes. In our case, task results contain, at maximum, two fields specifying a file path to the resulting files: either a log file or a file with statistics. Since we save only the paths, Redis data storage is sufficient for our needs. When *Flask* application wants to get an update regarding the task status, the server sends requests both to the broker and data storage to receive an updated status.

Additionally, we decompose the web application between multiple GitHub repositories. This separation allows us to ease maintenance and development by splitting different concepts (like back-end and front-end). We organised the structure of repositories in the following way:

- *Prosimos*¹³ encapsulates business logic of the simulation engine *Prosimos*. In addition, this repository allows us to use this simulation engine as a CLI tool and run it locally.
- *Prosimos Web Api*¹⁴ wraps the *Prosimos* CLI tool and exposes *Prosimos* functionalities as REST API endpoints. As a result, we are able to call simulation from the client (React application). Additionally, this repository includes setup for *Celery* task management service together with broker and data storage.
- *Prosimos Front End*¹⁵ contains *React* application, which concludes the front end part of the web application.
- *Prosimos Docker*¹⁶ summarises all previously noticed repositories in one Docker configuration file (*docker-compose.yml*). Later, we use this file to start all service instances as docker containers. So, we use this repository mainly for deployment versioning.

All of the repositories, except *Prosimos*, contain Dockerfile. Those files are necessary to create a Docker image. Once we have a Docker image, we create a Docker container, a running instance of an image.

¹²<https://redis.io/>

¹³<https://github.com/AutomatedProcessImprovement/Prosimos>

¹⁴<https://github.com/AutomatedProcessImprovement/prosimos-microservice>

¹⁵<https://github.com/AutomatedProcessImprovement/prosimos-frontend>

¹⁶<https://github.com/AutomatedProcessImprovement/prosimos-docker>

3.2 System Architecture

Prosimos 2.0 version of the tool is the logical continuation of the already existing solution Prosimos 1.0. While Prosimos 1.0 supports the limited number of elements in the standard BPMN 2.0 and can be run only via a console, Prosimos 2.0 introduces advanced BPMN elements and can be accessed via a browser and a console. In order to observe the tool's growth and changes, it is important to understand which part of the tool refers to which version. For this, we present the overall architecture of Prosimos 2.0 in Figure 3. We adopt this architectural view from our previously published paper [LPHD23] by highlighting the version in which each component is introduced.

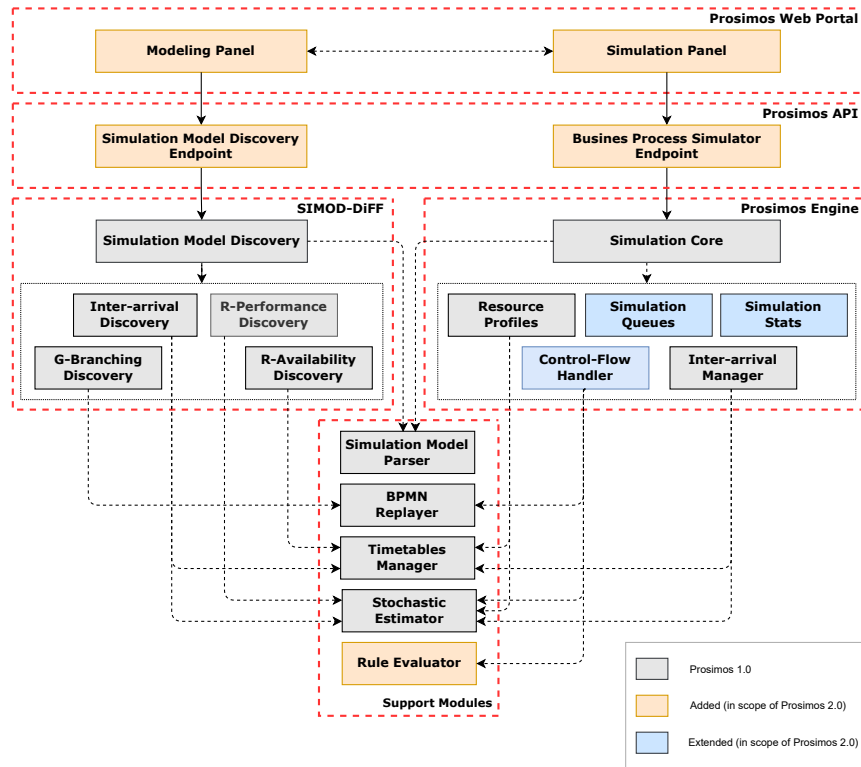


Figure 3. System architecture of Prosimos 2.0

Prosimos is logically structured into three layers, as shown in Figure 3. As legend states, components marked with orange colour on the figure are added (from scratch) in the scope of this thesis and those marked with blue - are extended based on the existing implementation from Prosimos 1.0. Both types of components were introduced as part of Prosimos 2.0. All other components, with a grey colour marking, belong to Prosimos 1.0.

The layer at the bottom, henceforth referred to as *Prosimos back-end*, consists of three groups of components labelled as SIMOD-DiFF, Prosimos Engine and Support

Modules. The Support Modules consists of a set of components containing supplementary functionalities shared by both the SIMOD-DiFF and Prosimos Engine. On the top-left of the back-end, the SIMOD-DiFF components discover the simulation parameters given a BPMN model and the corresponding event log written in XES or CSV, respectively. The Prosimos Engine components handle the business process simulation from a given model [LPHD23].

The middle layer of the architecture diagram, the Flask web application, exposes Prosimos REST API. Three endpoints are provided to the end user, grouped into the Simulation Model Discovery Endpoint and Business Process Simulator Endpoint. Table 1 describes the verbs, URIs, and actions of those endpoints. Although the REST API can span more specific endpoints, e.g., for interacting with the back-end to trigger the BPMN replayer or discover calendars for a given resource, for simplicity, we kept the API with the minimum operations required to discover simulation models and run simulations [LPHD23].

Verb	URI	Description
POST	/api/discover	Discovers the simulation parameters given a BPMN model and an event log
POST	/api/simulate	Performs the simulation from a given simulation model
GET	/api/results	Retrieves the event logs and metrics produced as result the simulation

Table 1. Prosimos REST API

On top of the architecture, the Modeling Panel describes a web interface for end-users to create, modify or discover (interacting with the Prosimos API) simulation models from event logs. On the right, the Simulation Panel allows users to run simulations and retrieve the resulting event logs and performance metrics. For further details about Prosimos back-end, we refer readers to our previous paper [LPHD23].

The web interface of the tool is divided into three main pages:

1. *Upload Page*, depicted in Figure 4, enables a user to provide two required inputs for the process simulation: business process model (.BPMN file) and simulation scenario (.json file).

There are three options for a starting point for setting up the simulation scenario:

- (a) *Create a simulation scenario manually* allows a user to start creating the scenario from scratch using the UI components instead of creating the file in a text editor. By using this option, a user benefits from an already prefilled list of activities or events or mapped options for a gateway.
- (b) *Upload a simulation scenario*. A user needs to provide a .json file in case of selecting this option. In order to be correctly parsed, this file should be

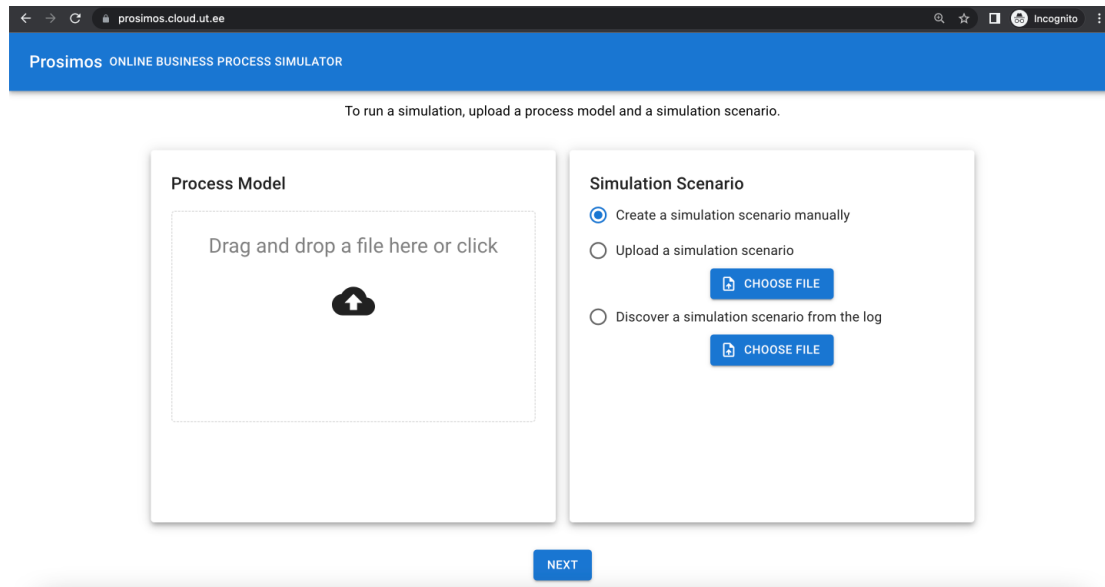


Figure 4. Prosimos upload page

previously exported from Prosimos itself. We do not support any other configurations inside this file except the one created for Prosimos. Additionally, a user is responsible for this file being matched with the business process file uploaded previously. The tool does not perform any cross-checks between the provided business process file and the simulation scenario file.

- (c) *Discover a simulation scenario from the log.* This option is used in case a user has a simulation log file of the process and wants the simulation scenario to be discovered from that file. There are some requirements for the column configuration in the file. The minimum essential columns should include case (unique identifier of the process case), activity (name of the executed activity), start (start date and time of the activity), end (end date and time of the activity), resource (name of the resource who executed the activity). The columns in the file should be equal to that list of keywords or contain them. The order of the columns does not matter.

2. *Scenario Overview Page*, shown in Figure 5, represents the content of the simulation scenario file and allows a user to modify it before running the simulation or exporting the scenario to be used later. We mentioned previously that .json contains seven sections. We combined the three first sections under *Scenario Specification* title on the UI. The rest four sections remain the same. Apart from sections from *Prosimos 1.0*, we have four additional tabs introduced as a part of *Prosimos 2.0*, namely *Intermediate Events*, *Batching*, *Case Attributes* and *Prioritisation*. So, the .json file is represented by nine sections on the UI. Some sections are

required to be filled in, among them are Case Creation, Resource Calendars, Resources, Resource Allocation. At the same time, other sections could be left empty. E.g., Branching Probabilities or Intermediate Events sections stay empty if there are no gateways or intermediate events in the BPMN model. If a user does not want to specify batch processing or case-based prioritisation, those sections also remain empty. The last tab, *Simulation Results*, is empty initially and populated once a user runs a simulation. There are a couple of buttons on the page with the following meaning:

- (a) *Start simulation*, first, validates a simulation scenario, e.g. whether all required sections were filled in or are in the valid, allowed range. Once the validation does not find any violations, a simulation starts executing. The time required for a simulation depends on several criteria, among which are model complexity and the number of instances to run. When a simulation finishes, a user is redirected to the *Simulation Results* view. This view is described in detail in a separate bullet point below.
- (b) *View model* visualises a BPMN model uploaded on the previous page in a separate browser window.
- (c) *Download as a .json* downloads the complete simulation scenario at the current stage of editing. This implies all changes introduced by a user are saved to the file. This option is useful if one wants to continue editing a simulation scenario or re-use a scenario later.

Prosimos ONLINE BUSINESS PROCESS SIMULATOR

[< UPLOAD NEW MODEL](#)
[START SIMULATION](#)
[VIEW MODEL](#)
[DOWNLOAD AS A .JSON](#)

[Case Creation](#)
[Resource Calendars](#)
[Resources](#)
[Resource Allocation](#)
[Branching Probabilities](#)
[Intermediate Events](#)
[Batching](#)
[Case Attributes](#)
[Prioritisation](#)
[Simulation Results](#)

Scenario specification

Total number of cases: 1000

Scenario start date and time: 04/06/2023 02:56 PM

Inter arrival time: expon

Loc (sec): 0

Scale (sec): 7200

Min (sec): 0

Max (sec): 100000

Arrival Time Calendar

Begin Day: Monday

End Day: Sunday

Begin Time: 00:00

End Time: 23:59

[+ NEW TIME PERIOD](#)

Figure 5. Prosimos scenario specification page

3. *Simulation Results* tab. When a simulation finishes, a user is redirected to the last, *Simulation Results*, tab. On this page, three tables with various key performance indicators (KPIs) are displayed: scenario statistics per metric, individual statistics per task and resource. Furthermore, there are two buttons:
 - (a) *Download stats* saves a file with data from those three aforementioned tables.
 - (b) *Download logs* button downloads a file with simulation logs of this current run.

Due to the fact that simulation is a stochastic process, each simulation run provides a unique set of results. Running the simulation for the second time results in overriding the data that was previously available to the user. There is no way of getting historical information about the specific simulation run. Additionally, we do not keep historical data on the server where the web application is deployed. We implemented the scheduled Celery task, which checks the server's file system and cleans files older than half an hour.

3.3 CI & CD pipelines

Maintaining the software refers to actions that we perform once we publish our software to end clients and they start using it. This might include fixing the bugs, improving the performance or introducing new features per user request. All of those activities might require a lot of costs unless a development team incrementally prepare the software to be maintained. Continuous integration (CI) & continuous delivery (CD) pipelines allow us to run build and test actions in one click. Additionally, it introduces software versioning which eases the process of deploying a new version of a software or moving to a previous one. Due to those advantages, CI & CD pipelines are one of those steps which simplify the maintenance of the app in the future.

With the usage of GitHub actions, we design our pipelines for *Prosimos*. GitHub actions are set up per repository. As *Prosimos* is distributed across multiple repositories, we set up each repository separately. However, there are some similarities between the setup. For example, publishing a docker image requires the same set of steps for both a Flask application and a React one.

The first pipeline is a CI one. The main goal of this step is to build and test a pushed version of the code. Figure 6 depicts how the continuous integration process looks like for the *Prosimos back-end* part. All GitHub actions require triggers, which start the execution of the action itself. For the CI pipeline, we specify three triggers: 1) creating a pull request, targeting the *main* branch; 2) pushing a commit to the *main* branch; 3) manually kicking off the CI process via GitHub UI. When one of the triggers is fired, the GitHub action is pushed to the queue for processing. The steps involved in the processing are defined by the user in *.yaml* file. This file is stored in the repository under the following

path: `.github/workflows/`¹⁷. The action is being executed on the server, which is called runner, provided by GitHub. Since the server does not include any configurations, the first step is to install *Python* together with a package manager *pip*. After that, we *install packages*, which are defined in a *package.json* file in the root folder. Once we have all dependencies installed, we run tests saved under the *testing_scripts* folder. CI pipeline for the React application includes the same steps with the difference that we install *Node.js* dependencies instead of *Python*. The *.yml* file is accessible under the same path but in a different repository¹⁸.

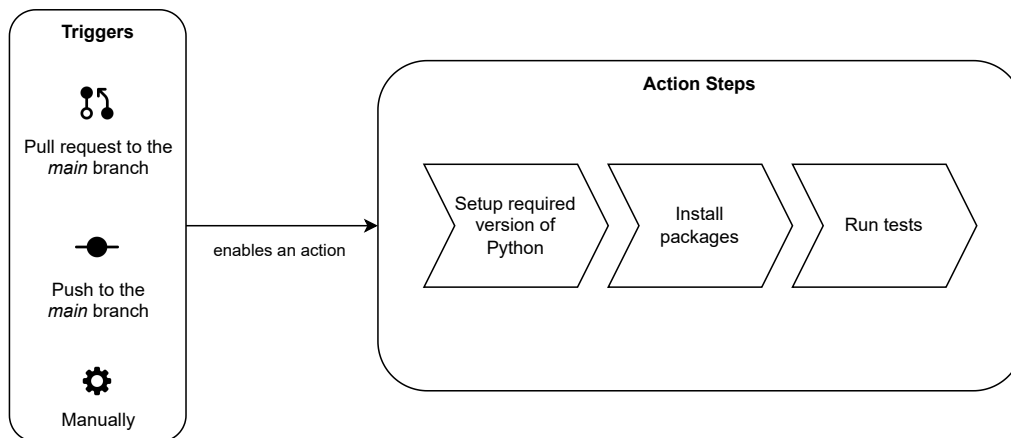


Figure 6. GitHub Actions responsible for CI

Another pipeline we introduce is the CD one. Its goal is to be able to release a new version at any moment. This results in releasing a new code version from time to time. Additionally, this allows us to move back to any version released in the past because we keep track of all releases. For this pipeline, there is the only allowed trigger is a new tag¹⁹ pushed to the GitHub. A tag is a pointer to any commit in the repository history. We use semantic versioning²⁰ to name tags and monitor all versions. The first step of the pipeline is extracting metadata, which contains a tag specifically. We use this tag for versioning Docker images. In order to be able to access Docker Hub²¹, where we store built images, we then log in to Docker Hub with personal credentials. After that, we build a Docker image and push the built image to the Docker hub. For *Prosimos back-end*, we have two applications: *Flask* and *Celery*. Once we are done with these steps, we have up-to-date docker images ready to be deployed on the server.

¹⁷<https://github.com/AutomatedProcessImprovement/Prosimos/tree/main/.github/workflows>

¹⁸<https://github.com/AutomatedProcessImprovement/prosimos-frontend/blob/main/.github/workflows/build.yml>

¹⁹<https://git-scm.com/book/en/v2/Git-Basics-Tagging>

²⁰<https://semver.org/>

²¹<https://hub.docker.com/>

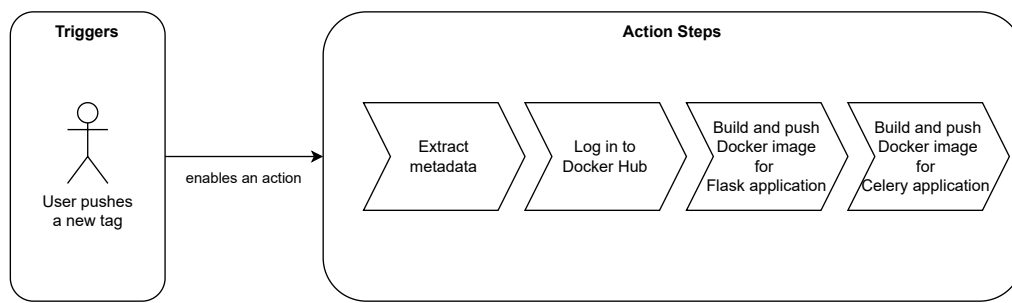


Figure 7. GitHub Actions responsible for CD

4 Simulation Enhancements

In this section, we focus on **RG1** and dive deep into newly introduced concepts of intermediate events, batch processing and case-based prioritisation in sections 4.1, 4.2, 4.3 accordingly. We describe those concepts individually in the following three subsections. The structure of each section remains the same. We start by eliciting high-level requirements. Then, we move into designing and implementing parts of the feature. After this, we present the implemented UI and describe its functionalities and user interactions with the tool. Finally, we delve into more technical details and describe the key changes introduced to the *Prosimos back-end*. Additionally, we provide pseudocodes for some of the algorithms for better visibility of the newly introduced concepts.

4.1 Intermediate Events

Requirements

- A user should be able to provide a model with an event-based gateway.
- System should be able to accept BPMN models with intermediate catch events, specifically message intermediate catch event, timer intermediate catch event, link intermediate catch event, and signal intermediate catch event.
- A user should be able to provide a distribution function per each event in a model.
- A user should receive a warning in case there are some validation errors in the provided data, e.g. one of the events does not have a specified distribution function.
- *Prosimos back-end* should allow providing a parameter of whether events should be included in the resulted log file or not. This parameter is a boolean value (two possible values: true or false) and, by default, set to False, meaning events are not added to the simulation log file.

Domain Design Introducing intermediate events and a gateway also requires being able to model them in regard to the simulation parameters. Our simulator needs to know when events are being executed cause we do not have any external interaction. *Timer* type, by default, specifies how much time we need to wait before the continuation of a process continuation. We use a duration distribution to be able to model this behaviour. However, *message* and *signal* are more complicated, and they do not have any duration component. Despite this, in this work, we allow to model them with the help of duration distribution, as well. This allows the simulator to know exactly when an event is executed. We name this time, allocated for the event execution, as *event duration* or *running time*.

As a result, our tool requires information about a distribution function for each event present in a BPMN model. Listing 1 presents how events should be described in the simulation scenario configuration file.

```
{
  ...
  "event_distribution": [
    {
      "event_id": "Event_18vjmc",
      "distribution_name": "fix",
      "distribution_params": [
        { "value": 900.0 },
        { "value": 0.0 },
        { "value": 1000.0 }
      ]
    }
  ]
  ...
}
```

Listing 1. Representation of event distribution in a simulation scenario file

We introduce a new section `event_distribution`, which contains an array of custom objects. This custom object consists of `event_id`, which should match a unique event id specified in a BPMN model, `distribution_name` and `distribution_params`. The last two properties specify the distribution function. With the help of this function, we specify the duration of the event (timer event) or when the event will be fired. When we have a distribution function defined, we can generate a value from this distribution, and this value is the one we use for a simulation. There are different libraries that help to receive this random number from the distribution function. In our case, we use *Scipy*²² library. There are several reasons for this decision. First of all, this library was already introduced in *Prosimos 1.0* so we can leverage it. Furthermore, this library contains a comprehensive number of supported functions compared to *NumPy* or *Pandas*. Additionally, *Scipy* library is well-supported and -documented.

The number of parameters the user is required to provide differs from the distribution function one uses. For example, if an event runs for a fixed period of time, e.g. 15 min, we use a `fix` function and specify 15 min in seconds (= 900 seconds) as a first parameter. The next two parameters of `distribution_params` defines the limit range, minimum (min) and maximum (max) boundaries, of generated values. If the generated value is outside of the provided range, it is discarded, and another one is generated. This happens until the moment the generated value lies within the specified range. The concept of maintaining generated values in a range was introduced for the distribution of activities' duration in *Prosimos 1.0*, and we use the same concept for events. In case the

²²<https://scipy.org/>

events' duration follows the normal distribution, we use norm distribution²³. We extract information about how many parameters we need to provide from the documentation, specifically from *rvs* method. For norm distribution, we need *loc* and *scale* parameters. Based on *Scipy* documentation²³, the location (*loc*) keyword specifies the mean and the scale (*scale*) keyword — the standard deviation. Additionally, our implementation requires two additional parameters: (*min*) and (*max*) values. As a result, *norm* requires four values provided as a *distribution_params* property.

UI Design Now, previously described concepts are implemented as UI components for an end-user. As we introduce a totally new section in a simulation scenario file, likewise, we add a new tab to the UI called *Intermediate Events*. Figure 8 illustrates the visual representation of the page. The tab lists existing events in a BPMN model. Consequently, an end-user accesses the page with already pre-filled data with identifiers of events. This is done in order to simplify the interactions with the tool. Once a user reaches this point, they should provide function distributions per every event in order to be able to run a simulation. If the data was not provided or was provided incorrectly, validation errors appear once a user clicks *Start simulation*. Validations that exist for this tab include 1) the presence of the data; 2) the data is in valid ranges: minimum value of 0 and no maximum value. Regarding the latter limitation, technically, there is a limitation of what memory allows us to save, e.g. value of 9007199254740991 is the highest and safest integer value that can be used in *TypeScript*. However, we assume that end-users do not utilise values that exceed the upper bound of the memory limitation.

Execution Semantics Regarding the introduction of the intermediate catch events to the *Prosimos back-end*, the following implementations take place. First of all, intermediate catch events describe processes or decisions which took place externally, meaning outside of the entities that execute a process. In this regard, intermediate events do not require an assigned resource in order to be executed. So when there is an event arriving at the control flow, we calculate when this event will be fired following the duration distribution. Once we know the date and time of an event firing, we put it in the priority queue, passing through this timestamp. Due to the simplicity of these changes, we do not provide its pseudocode.

In relation to event-based gateways, the implementation logic is described below. The function which decides which path the simulation executions takes is the main one responsible for the correct behaviour of this type of gateway. Algorithm 1 represents the logic implemented behind this. As input, we receive an array of outgoing flows from the event-based gateway. Elements of the array are represented by a data structure containing details about the intermediate event, among which are flow name and duration

²³<https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html#scipy.stats.norm>

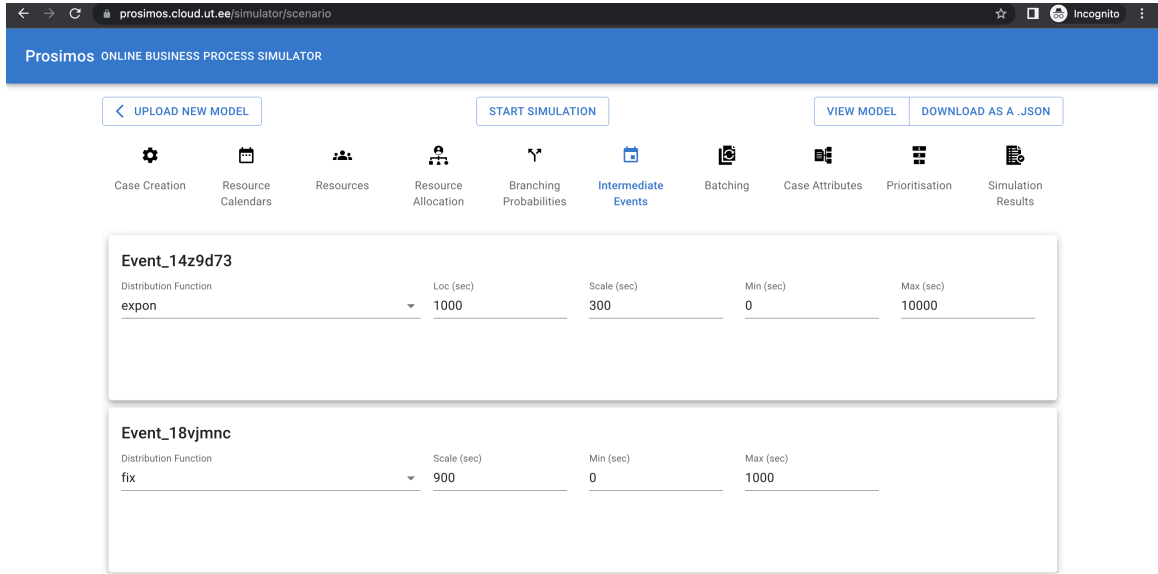


Figure 8. UI section, describing intermediate catch events

distribution function. Lines 3-5 compute the duration of each event that is outgoing from the gateway. After, we find the index of the lowest duration by executing function `GetIndexOfMinValue`. As an execution result, we return the information about the selected intermediate event (line 7).

Algorithm 1: Event-based Gateway Decision Algorithm

```

1 function GetEventGatewayChoice (outgoing_flows: []):
2   all_outgoing_flows_dur  $\leftarrow$  [];
3   for flow_index = 0 to length(outgoing_flows) do
4     all_outgoing_flows_dur[flow_index]  $\leftarrow$ 
       GetCurrentFlowDuration(outgoing_flows[flow_index]);
5   end
6   min_dur_flow_index = GetIndexOfMinValue(all_outgoing_flows_dur);
7   return outgoing_flows[min_dur_flow_index];
8 end

```

4.2 Batch Processing

Requirements Batching is the next feature we introduce as a part of this work. We want to allow the simulation of activities executed not individually but in a group, meaning a

batch, with others. In order to design and implement batching, we form a list of initial requirements:

- A user should be able to define batching per individual task.
- System should support batching for multiple tasks.
- A user should describe batching, providing batching type, batching probability, duration scaling and firing rules.
- System should support two batching types: parallel and sequential.
- Allowed range of batching probability is from 0 to 1.
- Task should be batched only when the firing rule assigned to the task is true.
- System should support five types of rules: by the size of a batch, the hour of the day or day of the week, and waiting times. Additionally, waiting times contain two separate types: waiting time since the enablement of the first activity in a batch and since the enablement of the last activity in a batch.
- In regard to simulation, a formed batch is executed by one and only one resource.

Domain Design Having high-level requirements defined, we create a domain diagram to visualise relationships between existing and newly introduced concepts. Figure 9 depicts those relationships. *BPMN Activity* represents activity from a BPMN model. This is an existing entity from *Prosimos 1.0*, and it contains numerous attributes. However, the only attribute which interests us is *taskId*, a unique identifier of a task. An activity can or can not have an assigned batch setup. The cardinality of the relationship *attached to* shows that an activity can be attached to 0 or 1 batch setup. This comes from the requirements as this relationship is not mandatory, and an end-user is not required to set up batching for all existing tasks. We describe batching for the specific activity by *BatchSetup* entity. It includes batching type, how much (in percentage) instances of tasks are batched, and how task duration changes in accordance with the number of tasks in a batch. We also need to know when a batch is enabled in order to start processing it. For this, we define *OrBatchingRule*, which contains a list of logical OR rules. OR rule, in turn, contains a list of logical AND rules. Each item inside the AND rules list represents the smallest atomic entity, *BatchingRule*. We use the same approach of a chain of rules for designing another feature, case-based prioritisation. Consequently, a deeper description of this *OR rule* \rightarrow *AND rule* \rightarrow *atomic rule* chain is described later when we delve into a domain design of a case-based prioritisation.

Moving ahead, the next concept we describe is the *BatchingRule* entity. By using the word *atomic* previously, we emphasise that this is the smallest possible in this domain

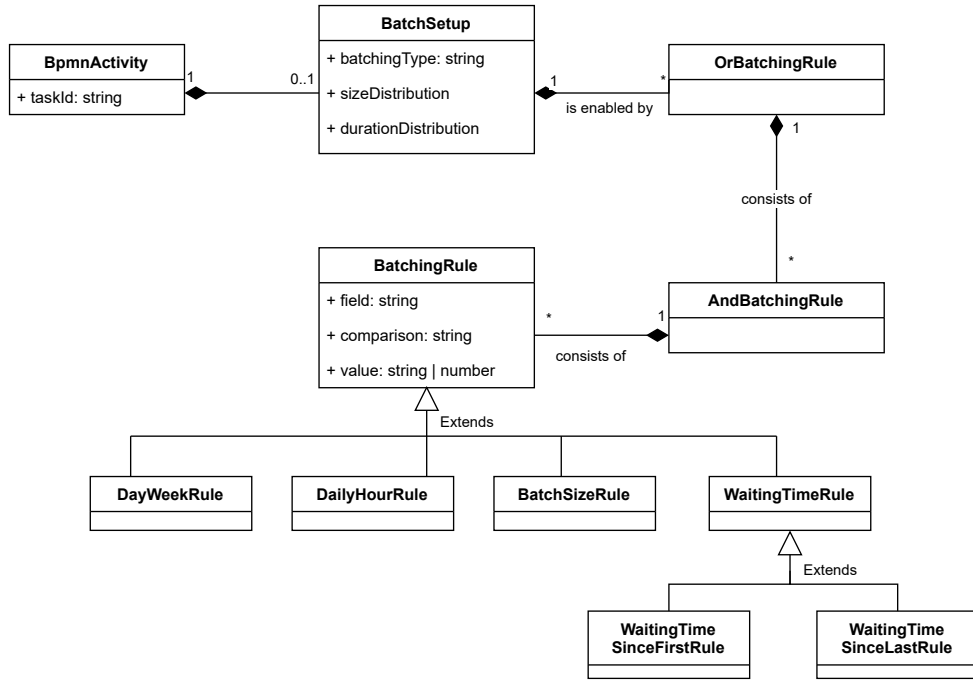


Figure 9. Domain model of batching

entity that can not be divided into smaller pieces. Also, this entity is the one around which we build firing rules which enable batch execution. *BatchingRule* contains three attributes: *field*, *comparison*, *value*. We provide details on what values are allowed per each of these attributes later in this section. Since we have a limited number of allowed values for *field*, we depict that by introducing the generalization relationship between *BatchingRule* and derived entities from that, namely *DayWeekRule*, *DailyHourRule*, *BatchSizeRule*, *WaitingTimeRule*. Additionally, we split *WaitingTimeRule* entity based on a starting point for the counter of waiting time. As a result, we form two new child entities *WaitingTimeSinceFirstRule* and *WaitingTimeSinceLastRule*.

Based on the description of the domain, we derive a JSON schema. Listing 2 provides JSON data which complies with the designed schema. As multiple tasks can be batched, root element *batch_processing* contains an array of items. Each item represents an object and consists of the following fields:

- *task_id* represents the task to which we apply the batching. Here, we reference the task identifier saved in the BPMN model.
- *type* specifies how tasks are executed inside the batch. There are two possible options: "Parallel" or "Sequential". Parallel describes a situation when a resource executes all tasks together and there is no specified logic on how a resource switches between them. Consequently, all tasks present in a batch have the

same start and end time in a resulting log file in our implementation. Sequential type refers to a case when a resource executes tasks in a batch one after another. In this case, a resource devotes his time to one task at a time and does not shift focus to other tasks from the batch. This implies that a resource starts executing the next task only after finishing executing the previous one.

- *size_distrib* defines the percentage of tasks being executed individually or batched. The provided example in Listing 2 states that all tasks (where `id = task_id_1`) are batched. In case one wants to define that, for example, 20% of tasks are executed individually and the rest, 80%, - in batches, we need to have two items in the list. The resulting setup should look like this: [{"key" : "1", "value" : 0.2}, {"key" : "2", "value" : 0.8}]. The field's structure is designed in a complex way intentionally. The second usage of this field is configuring a batching rule based on the probability of the number of tasks. So we can represent the following scenario: *In 30% of cases, the batch happens when there are four tasks in a batch. All other batches (70%) happens when there were only two tasks in a batch.* In this case, the fields' value is structured as follows: [{"key" : "2", "value" : 0.7}, {"key" : "4", "value" : 0.3}]. We treat this field as a batching rule only in case *firing_rules* is empty.
- *duration_distrib* describes how the duration of tasks inside a batch is impacted by the number of tasks in a batch. One of the examples we provided before refers to improved performance after batch introduction. Therefore, this section enables users to adjust the duration of the tasks by a scale factor. The example from Listing 2 states that the duration of individual tasks inside a batch will be reduced by 0.2 for batches with a size of 3 or more items. The structure of this field also allows more sophisticated scenarios, combining different batch sizes. For example, we can define that we reduce the performance by 0.2 for batches of sizes 3, 4, 5 and by 0.25 for batches with 5 or more tasks inside.
- *firing_rules* specifies a condition under which a batch is enabled. The structure follows the same idea of combining OR and AND conditions under one rule as we describe later for prioritisation rules. Accordingly, the rule from Listing 2 is read like *(daily_hour < "12" AND week_day = "Friday") OR (size >= 4)*. Compared to prioritisation rules, batching rules introduce a different approach to atomic rules. We can define five types of rules for batching: *daily_hour*, *week_day*, *size*, *ready_wt*, *large_wt*. All those names are reserved for *attribute* field. We describe those types separately by providing information on what they mean and allowed values.
 - *daily_hour* specifies hour of the day. The allowed range of this value is from 0 to 23. We do not provide any opportunity to specify the time


```

{
  ...
  "batch_processing": [ {
    "task_id": "task_id_1",
    "type": "Parallel",
    "size_distrib": [ { "key": "2", "value": 1 } ],
    "duration_distrib": [ { "key": "3", "value": 0.8 } ],
    "firing_rules": [
      [
        {
          "attribute": "daily_hour",
          "comparison": "<",
          "value": "12"
        },
        {
          "attribute": "week_day",
          "comparison": "=",
          "value": "Friday"
        }
      ],
      [
        {
          "attribute": "size",
          "comparison": ">=",
          "value": 4
        }
      ]
    ]
  } ]
  ...
}

```

Listing 2. Representation of task batching in a simulation scenario file

with exact minutes or seconds. An end-user is required to specify the time only by specifying a clock hour. The allowed list of operators for this type is $< \leq = > \geq$. We name this list `relational operators` and use it throughout the paper. If a user wants to define a range during the day, a solution is to use multiple statements under the same AND rule. For example, *daily_hour* ≥ 13 and *daily_hour* ≤ 18 describes a range from 1 pm to 6 pm, including.

- `week_day` represents the day of the week when the rule is enabled. The number of allowed values for this type is limited by the number of days in the week. This means we can select a value from a list of seven weekdays. The format of a weekday is a capitalised day of the week, for example, "Friday". The only allowed comparison operator is $=$. We encourage the usage of OR and AND combinations to design more complex enabling rules, such as batches being enabled on Mondays and Fridays.
- `size` is used to limit the number of items per batch. As a value, we provide any integer number (following the programming language limits). The list of allowed operators contains items from Boolean algebra and is the same as used for `daily_hour` type.
- `large_wt` stands for a waiting time (in seconds) since the first enabled time of activity in a batch. Figure 10 depicts the logic behind the term. We use [LMCCD22] as a starting point for this figure and extend it by adding an explanation about `large_wt` (same as *WTlarge*) and `ready_wt` (same as *WTready*). In general, the figure visualises a scenario of how a batch is formed in regard to the time perspective. We split the process into two parts *batch accumulation* when we wait for activities to satisfy the enabled rule, and *batch processing* when we already have activities selected for a batch, a rule enabled the batch and resource executes activities. For `large_wt`, we start our time counter when the first activity is enabled in the batch, meaning at *tenabledC1* point of time, and the counter lasts till the moment the batch is enabled. This type describes an upper boundary on how long a user is willing to wait for a batch to be formed. `Relational operators` defines the range of allowed comparison operators for this type.
- `ready_wt` defines a waiting time (in seconds) since the last enabled time of activity in a batch. Compared to `large_wt`, we update the value of this metric every time new activity gets added to a batch. With `large_wt`, the value is set up only once when the first task in a batch is enabled. Figure 10 illustrates the difference of the starting point for `large_wt` and `ready_wt`. It also demonstrates that the finishing point for both of them is the same, and it is a point in time when a batch is enabled. The list of an allowed operators in `comparison` field is the same as used for `large_wt`.

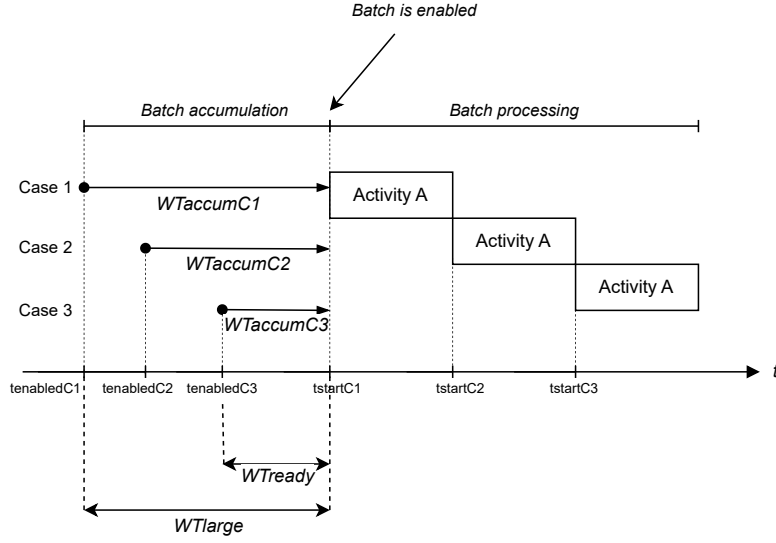


Figure 10. WT_{ready} and WT_{large} in regard to the time axis, including other waiting times

UI Design Next step of the feature introduction is designing and implementing the user interface for a new section in a simulation scenario. Figure 11 illustrates the user interface created for task batching. We present results as two screenshots combined together due to the dimensions of the UI components. Figure 11b presents a batch rules' visualisation, and Figure 11a - the rest of the required attributes, which we explain later on.

When a user initially opens the *Batching* section, there is a list of all batched tasks with collapsed details (if a previous setup for batching exists). This starting view provides an overview of how many tasks are batched in the current simulation scenario. Once an end-user expands the details of one of the tasks, we see various sections from a .json file with a simulation scenario. We describe those components one by one:

- *Batching Type* shows a type of batching: either *sequential* or *parallel*. Those options are presented in the form of a dropdown for a better user experience allowing a user to choose an option instead of typing the whole word.
- *Batching Probability* describes the probability of whether the task is batched or not and maps to *size_distrib* field in a .json file. This component somehow differs from the JSON schema. The user does not have an opportunity to specify the probability of each batch size. This is done to simplify the UI. Therefore, an end-user provides a probability for "key" : "2". In case we want all tasks with this *id* to be executed alone, we provide 0 as a value for this field.

Prosimos ONLINE BUSINESS PROCESS SIMULATOR

UPLOAD NEW MODEL START SIMULATION VIEW MODEL DOWNLOAD AS A .JSON

Case Creation Resource Calendars Resources Resource Allocation Branching Probabilities Intermediate Events **Batching** Case Attributes Prioritisation Simulation Results

+ NEW TASK BATCHING

Task

Task: D

Batching Type: Parallel

Batching Probability
Probability (from 0 to 1): 1

Duration Scaling
Batch Size: 3 Scale Factor: 0.8

Firing Rules

(a)

Prosimos ONLINE BUSINESS PROCESS SIMULATOR

UPLOAD NEW MODEL START SIMULATION VIEW MODEL DOWNLOAD AS A .JSON

Case Creation Resource Calendars Resources Resource Allocation Branching Probabilities Intermediate Events **Batching** Case Attributes Prioritisation Simulation Results

+ NEW TASK BATCHING

Task

OR

AND

Field: Day of the week Operator: Equals Value: Friday

Field: Hour of the day Operator: Between Value: 0 Value: 12

AND

Field: Batch size Operator: Greater Than or Equal To Value: 4

(b)

Figure 11. UI sections, defining batching in regard to (a) general set of attributes, (b) firing batching rules

- *Duration Scaling* presents *duration_distrib* field from a .json file. This component shows a list of items with an opportunity to add a row (by clicking + button on the top right) or delete a row (by clicking a *bin* button near a specific row). We have also added validation, and an end-user is not allowed to provide rows with duplicated keys. The value of a scale factor should be in the range from 0 to infinity, where infinity here refers to the highest possible number allowed in *Typescript*.
- *Firing Rules* describe *when* the batching is enabled. The designed UI component has OR and AND keywords encoded and visualised so that it helps an end-user to design a needed rule. The outer array of items, OR condition, always stays as a root of the configuration. Following this, an end-user is able to add one or multiple AND conditions with atomic rules inside. Each atomic rule is represented by three inputs: *Field*, *Operator* and *Value*. *Field* is represented by a drop-down menu with five predefined options based on the types we described earlier. Based on a selected *field*, the other two inputs change their initial state and behaviour. So we describe each option of a menu separately, and we follow the same order used for the explanation of JSON schema of types of rules:
 - *Hour of the day* contains two possible operators: *Equals* and *Between*. *Value* changes its appearance based on a selected operator. When *Equals* is selected, an end-user is required to provide only one integer value from the allowed range from 0 to 23. With *Between* operator selected, a user needs to provide two values, lower and higher boundary of the range, in the mentioned order. On user input, the UI components validate whether a first value, lower boundary, is lower than a second one, higher boundary. After a user specifies a range of values, we transform the provided rule to the structure supported by our back-end. We describe the transformation rules later on.
 - *Day of the week* has only one allowed operator *Equals* as per requirements. *Value* input is depicted by a drop-down menu, as well, containing seven options, one per each day of the week.
 - *Batch size* has all five operators to which we refer as relational operators. As there might be situations when we do not want to have an upper limit for a *batch size* type, we have not introduced *Between* operator instead of raw operators. So a user is free to choose whether one wants to have a rule with one value or a range of them. For the latter one, a user needs to add two rules with a *batch size* type.
 - *Time since first* and *Time since last* differ only by a starting point of a time counter. The other logic for those two types is similar, so we describe them together. For these types, the allowed list of operators includes: *Equals* and *Between*. Both of those operators follow the same strategy as described for *Hour*

of the day type. The only difference is the units of measurement since time is measured in seconds in our case. As we want to limit users from providing an infinite range for a waiting time, we introduced *Between* operator instead of allowing users to select an operator from a list of relational operators.

It can be seen that three types out of five introduce *Between* operator. This new option is designed to simplify the interaction with a tool and make rules more readable for an end-user. However, this design also requires the introduction of the functional logic, which transforms a *Between* operator to the list of relational operators supported by the back-end. The following examples capture the main idea behind the logic:

1. *ready_wt between 10 and 100* \Rightarrow *ready_wt \geq 10 and ready_wt \leq 100* when a lower boundary is not equal to 0 and an upper one is not equal to the reserved word *inf*, we have both boundaries present. As a result, a new transformed rule contains two parts combined with an *and* condition.
2. *ready_wt between 0 and 100* \Rightarrow *ready_wt \leq 100* when a lower boundary is equal to 0, we can disregard it, as 0 is the lowest possible number when we talk about real-time duration. Consequently, a new rule contains only an upper edge in the rule and a lower boundary is enforced by default.
3. *ready_wt between 100 and inf* \Rightarrow *ready_wt \geq 100* when an upper boundary is equal to the reserved word *inf*, we treat this like an absence of a lower boundary. Therefore, we omit it and a resulting rule contains only a lower boundary. This type of setup is not allowed for batching and is used only with prioritisation rules which we discuss later.

Execution Semantics In regard to *Prosimos back-end*, development of this feature is split into multiple stages: 1) parsing a batching section from a simulation scenario, 2) evaluating whether a firing rule is true and hence whether a batch is ready to be executed, 3) calculating a size of an enabled batch and enabled time of a batch, 4) tracking batch enablement during the whole process of a simulation process.

Transforming a JSON section into classes and validating them is a trivial task, so we do not provide any details on this part. One can check the source codes of the project; specifically the part implemented for batch processing²⁴, and derive the parsing logic from there.

We formulate rules for both batch processing and case-based prioritisation using the same approach. As a result, their parsing and evaluation have some common logic behind them. Specifically, the OR and AND rules evaluation follows a similar pseudocode

²⁴https://github.com/AutomatedProcessImprovement/Prosimos/blob/main/bpdf_r_simulation_engine/batch_processing.py

described in Algorithm 4. We provide a line-by-line explanation of the algorithm later in the next section. At a high-level overview, the pseudocode evaluates whether an OR rule, supplied as an input, is true or false. We achieve this by looping over all subrules and applying boolean algebra of OR and AND boolean expressions. One of the distinguishing features is what we return as a result of function execution. In the case of batching, we return just a truth value of an OR rule, while we return the priority level of a truthful rule in case of prioritisation.

After evaluating whether a firing rule is True and enables batching, we need to calculate the number of activities in a batch and an enablement time of a batch. Both of these values depend on the setup of firing rules provided by an end-user. Algorithm 2 depicts the main logic behind the calculation. This function is a part of `AndFiringRule` class, representing `AndBatchingRule` entity from the domain model depicted in Figure 9. `AndFiringRule` class consists of a list of atomic rules, referred to as `and_rules_list`, which is the first input parameter in Algorithm 2. Two other input parameters we receive are `current_batch_size`, the number of activities currently waiting for batch execution, and `batch_info`, an object containing information about a batch setup, among which are waiting time and enabled time of each activity waiting for batching. First, in line 2, we assign the initial value of `batch_size` to the maximum possible integer number, and later we reduce this number during our iterations. Lines 2 and 3 temporarily save the current point of time of simulation execution. We use this time to calculate the waiting times of activities. Then, we calculate the size and enabled time of a batch per each atomic rule in an AND rule. As `batch_size` type of rule limits the size of a batch, we put this rule as the last one in a list during parsing. By this, we guarantee that the resulting number follows the size rule. In line 5, we start by calculating those two values taking into account the waiting times. `is_time_forced`, in line 6, is a boolean value which tracks whether `daily_hour` rule enforces the enabled time of a batch. After that assignment, we loop over all present rules to see whether we need to adjust the resulting values, namely `batch_size` and `enabled_time`. `subrule` represents an atomic rule and contains three properties: `field`, `comparison_operator` and `value`. Those properties match with the list of entity fields. We create a separate function per each type of rule to calculate new values and after that, we compare them with the ones we had from the previous iteration. The first if-statement in line 9 checks whether a rule from the current iteration is of type `batch_size`. If it is, we call a function `GetFiringBatchInfoByBatchSize` and pass all currently calculated information. The function calculates new values based just only on a `batch_size` rule. Afterwards, we calculate whether this value satisfies the previous value, meaning `current_batch_size`. Satisfying means lying inside the range: if $current_batch_size \geq batch_size$, we split and execute multiple batches taking into consideration the upper boundary of `batch_size`. Otherwise, we return 0 as we do not have enough items to proceed with a batch execution. Returning back to the algorithm, we continue with another type of rule in line 13, which is `week_day`.

The same happens here: if the rule is of type `week_day`, we call the function of this type, `GetFiringBatchInfoByWeekday`, and calculate new returning values. We also introduce here a short circuit. This is done in order not to consume resources when we already know that this function does not return a satisfying result. A batch is an execution of two or more activities together. Consequently, if one of the rules returns a batch size of 1, we do not qualify this as a batch and skip execution till the next point in time. Lines 15-17 and 27-29 represent this short circuit. Line 18 changes the value of `is_time_forced` to `True`. This is used in case we additionally have `daily_hour` rule. Value of `is_time_forced` helps us know whether we are limited in days of the week when calculating the enabled time or not. The next type of rule we check is `daily_hour`. In line 21, `only_one_date` is a copy of `is_time_forced`. In case `is_time_forced` is `True`, we update the `curr_enabled_at`, the current point of time, to the enabled time calculated based on a `week_day` rule. We do this in order not to lose this date enforced by another part of an AND rule. The statement in line 30 is equal to `True` only in case we have both `week_day` and `daily_hour` together. In this case, the result of `daily_hour` overwrites the previous result from `week_day` in line 31. There might also be situations when the `daily_hour` rule exists without a `week_day`. In this case, the final result still needs to enforce the hour range, so we change the value of `is_time_forced` for this. At the end of each iteration, we check whether the `batch_size` enforced by this rule is lower than the previous one. If yes, we take the lower one as a result of this iteration and continue with the next rule from a list. Lines 35-37 skip calculations for `ready_wt` and `large_wt` since we do this in line 5. Lines 42-44 handle situations when the received result does not satisfy one of the requirements, and no batch should be executed. We do not proceed with batch execution when 1) no iterations are made or 0 as a resulted size of the batch, 2) `None` as an `enabled_time`, 3) `enabled_time` is in the future (they are executed later).

Algorithm 2: Calculate the batch size and enabled time of the batch

```
1 function GetBatchSizeAndEnabledTime (and_rules_list, current_batch_size, batch_info):
2   batch_size  $\leftarrow$  sys.maxsize;
3   initial_curr_enabled_at  $\leftarrow$  batch_info["curr_enabled_at"];
4   enabled_time  $\leftarrow$  initial_curr_enabled_at;
5   batch_size, enabled_time  $\leftarrow$ 
      GetFiringBatchInfoBySinceFirstAndLastWtRule(batch_info, batch_size, enabled_time);
6   is_time_forced  $\leftarrow$  false;
7   for subrule in self.rules do
8     curr_size  $\leftarrow$  0;
9     if subrule.IsBatchSizeRule () then
10      return GetFiringBatchInfoByBatchSize(subrule, batch_info, current_batch_size, batch_size,
11      enabled_time, initial_curr_enabled_at, is_time_forced);
12    end
13    else if subrule.IsWeekdayRule () then
14      curr_size, enabled_time  $\leftarrow$  subrule.GetFiringBatchInfoByWeekday(batch_info);
15      if curr_size < 2 then
16        | batch_size  $\leftarrow$  0; break;
17      end
18      is_time_forced  $\leftarrow$  true;
19    end
20    else if subrule.IsDailyHourRule () then
21      only_one_date  $\leftarrow$  false;
22      if is_time_forced then
23        | batch_info["curr_enabled_at"] = enabled_time;
24        | only_one_date = true
25      end
26      curr_size, enabled_time  $\leftarrow$ 
        subrule.GetFiringBatchInfoByDailyHour(batch_info, only_one_date);
27      if curr_size < 2 then
28        | batch_size  $\leftarrow$  0; break;
29      end
30      if is_time_forced then
31        | batch_size  $\leftarrow$  curr_size
32      end
33      is_time_forced  $\leftarrow$  true if (enabled_time! = None and enabled_time.time() ==
        subrule.value) else false;
34    end
35    else if subrule.IsSinceLastWtRule () or subrule.IsSinceFirstWtRule () then
36      | continue;
37    end
38    if curr_size < batch_size then
39      | batch_size = curr_size;
40    end
41  end
42  if batch_size in [sys.maxsize, 0] or enabled_time == None or
    enabled_time > initial_curr_enabled_at then
43    | return 0, None
44  end
45  return batch_size, enabled_time
46 end
```

The complexity of the fourth part arises from two types of rules associated with waiting times: *large_wt* and *ready_wt*. Their presence requires us to track at which point of time we are currently during the execution so that we do not miss the point when we need to enable a batch. For this, we created a number of functions which continuously verify the time and the rule enablement. Additionally, there is another edge case, meaning a rare situation that could or could not happen to an end-user. What happens if the provided rule is not satisfied during a process simulation due to the conflict in those rules? For example, a user defines the rule with *batch_size* ≥ 7 and *large_wt* > 7200 and *large_wt* < 9000 . We call this rule a complex one because it contains more than one atomic rule. The rule "says" that we need to collect at least seven items per batch and wait from two to two and a half hours. It might be possible that we received only five activities waiting for a batch execution during the period of two and a half hours. And once we surpass a limit of two and a half hours, there is no possible way that this rule will be fulfilled later on. In light of this situation, we added additional logic to handle it. In case a complex rule containing waiting time atomic rules (either *large_wt* or *ready_wt*) surpasses the duration boundaries, we evaluate this rule as a true one. This means that we enable a firing rule immediately after waiting for two and a half hours in the case of our previous example. It also might happen that we still have open batches waiting to be enabled by a rule at the end of a simulation. An open batch means one which already has some items inside it but is not enabled yet by a firing rule. For example, if the rule requires us to have three activities in a batch and right now we only have two activities, a batch with those two items is called an "open" batch. We handle those types of scenarios by enabling all "open" batches at the end of the simulation. In this case, the end of a simulation is marked by the last case started. By tackling open batches, we guarantee that there are no unfinished case instances as a result of a process simulation.

4.3 Case-based Prioritisation

Requirements In *Prosimos 1.0*, activities are being executed based on their enabled time (taking that the resource that executes the activity is available). However, this behaviour might be different in real life. For example, the activity can be prioritised by the resource executing the activity, or some specific attributes of the process instance can define the priority of the activity execution. In line with the latter example, we want to allow users to introduce prioritisation by *case_attributes*.

Additionally, if we add support for case-based prioritisation, we first need to support case attribute generation. This feature implies that a user is able to provide a list of attributes which will be calculated per every case instance during a process simulation. Examples of those case attributes could include: *client_type* or *loan_amount* in some loan application process. *Prosimos 1.0* do not support that functionality, so we introduce those changes in *Prosimos 2.0*.

The following list of items describes requirements that should be satisfied after the implementation of case attribute generation and case-based prioritisation:

- A user should be able to add a case attribute to a simulation scenario by specifying a name for a case attribute and how the value should be generated for this attribute.
- All case attributes should have a unique name.
- There should be two allowed types of case attributes: discrete and continuous.
- For discrete case attributes, a user should be able to provide an array of possible values together with the probability of each of those values.
- For continuous case attributes, a user should be able to specify a function distribution for the attribute's value.
- A user should be able to specify the rules and their appropriate priority level.
- The lowest priority level that can be used is one, and there is no upper limit. The upper limit here is restricted by programming languages limitation - *Typescript* and *Python*.
- The lower the number of a priority is - the higher this priority is treated, e.g. 1 - is the highest possible priority.
- The priority levels should be unique; no duplication is allowed.
- UI components should have hints for the discrete case attributes since possible values are limited in this case.

Domain Design Since this feature is split into two parts, case attributes generation and case-based prioritisation, we split this section into two paragraphs, as well.

Case Attributes Figure 12 presents domain model of case attributes. It consists of two main entities `AllCaseAttributes` and generalised `CaseAttribute`. `AllCaseAttributes` describes a list of all case attributes, while `CaseAttribute` - individual case attribute. Those individual case attributes are described by two fields: name, name of the case attribute, and `valueConfig`, description of how values should be generated for this case attribute. Entity `CaseAttribute` also generalises `DiscreteCaseAttribute` and `ContinuousCaseAttribute`. Discrete and continuous types of case attributes are the only allowed type and we use `CaseAttribute` as a superclass which shares the common logic of both case attributes. Child entities, `DiscreteCaseAttribute` and `ContinuousCaseAttribute`, contain information specific to only those types.

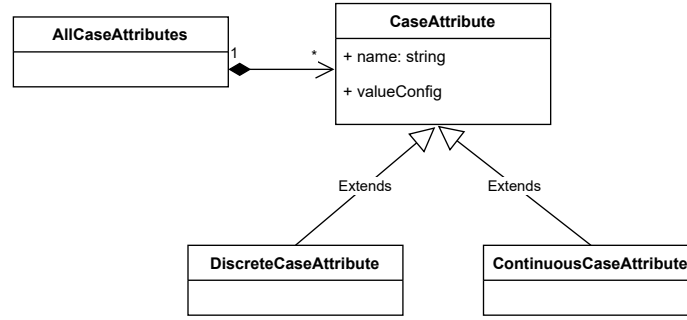


Figure 12. Domain model of case attributes

After the design of the domain, we continue with designing how the case attributes described in a simulation scenario file. Listing 3 presents two examples of different types of case attributes, where *client_type* is of type discrete and *loan_amount* represents continuous case attribute.

Both types of case attributes have a fixed number of fields: name, type and values. name describes the name of the specific case attribute, and type - type with only available options of continuous and discrete. While the first two fields are pretty straightforward, the structure of the latter one differs based on the case attribute's type.

Continuous case attributes are described with the help of function distribution. Here, we use the same approach as for designing the function distribution of the intermediate events. As a result, we provide *distribution_name*, a name of the function, e.g. *fix*, and *distribution_params*, a number of numerical parameters which varies based on the used distribution function.

As for discrete type, the name is already suggesting that this case attribute can only take as input discrete values. Consequently, it is required to provide a list of possible options for this case attribute. In order to be able to match a value to a specific case instance, we also need to define the probability of each option being selected. Internally, we use normal distribution for selecting the next case attribute value. Listing 3 describes an example of *client_type*. In this specific case, the case attribute value can be equal to one of the available values: either *REGULAR* with a probability of selection of 80% or *BUSINESS* with a probability of selection of 20%. This results in a situation when 20% of simulated case instances have *BUSINESS* assigned as a *client_type*, while other 80% - *REGULAR*.

Case-Based Prioritisation Figure 13 depicts the domain diagram designed for case-based prioritisation. It consists of five main entities, namely *AllPrioritisationItems*, *Prioritisation Rule*, *OrRule*, *AndRule*, *Rule*. We use a composition link for all relationships in the diagram. This is followed by a use case that it does not make sense to have any of the domain elements individually. For example, having *AndRule* without

```

{
  ...
  "case_attributes": [
    {
      "name": "client_type",
      "type": "discrete",
      "values": [
        { "key": "REGULAR", "value": 0.8 },
        { "key": "BUSINESS", "value": 0.2 }
      ]
    },
    {
      "name": "loan_amount",
      "type": "continuous",
      "values": {
        "distribution_name": "fix",
        "distribution_params": [
          { "value": 240 },
          { "value": 0 },
          { "value": 1000 }
        ]
      }
    }
  ],
  ...
}

```

Listing 3. Representation of case attributes in a simulation scenario file

a parent of OrRule element is not contributing to any of the specified requirements. This means AndRule should exist only together in composition with OrRule. As a result, deleting OrRule results in deleting the underlying AndRule, as well (we should not be left with an orphaned AndRule).

AllPrioritisationItems entity holds information about multiple priority levels, which are defined by a user in a simulation scenario. An individual priority level is described as a PrioritisationRule entity. Throughout the paper, we use the terms *prioritisation rule* and *priority rule* interchangeably. Each PrioritisationRule has priority property which defines the level of priority of this rule. PrioritisationRule is enabled by OrRule, meaning a current priority rule is applied to the process case only if OrRule is true. In general, OrRule represents a logical rule with a set of conditions. This rule is true if one of the conditions inside this rule is true. Each of the conditions is represented as AndRule. This entity, AndRule, also denotes a logical rule with a list of conditions inside. However, this time the rule of type AND is equal to true only in case all of the conditions are equal to true.

As a final point, the smallest atom of the presented diagram is Rule. This entity,

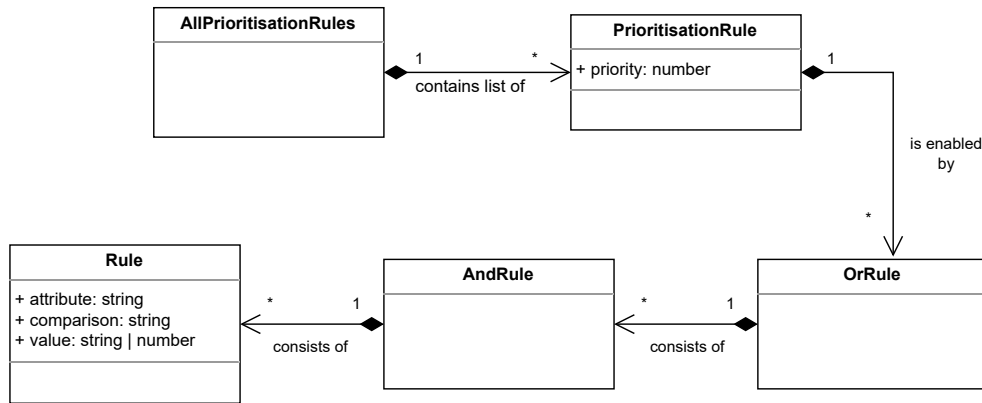


Figure 13. Domain model of case-based prioritisation

Rule, outlines the atomic condition statement, which in the end, is being evaluated. Evaluating a condition statement means checking whether it is truthy or false. Figure 14 presents examples of rules that could be defined by a user. The first one says that case attribute *client_type* should be equal to *BUSINESS*. When this statement holds true, the condition statement is true. If *client_type* is equal to a value different from *BUSINESS*, the evaluation of this condition is false. The second statement is an example of the continuous rule. This specific condition is true only when case attribute *loan_amount* is less than or equal to 1000.0.

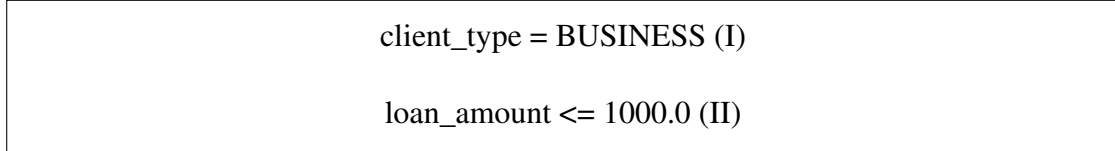


Figure 14. Examples of: I) discrete rule, II) continuous rule

Based on the allowed types of case attributes, discrete and continuous, there are also two same allowed types of rules. In terms of case-based prioritisation, the difference between those two rules lies in the allowed comparison operators and values. For discrete rules, the only allowed operator is $=$. If an end-user wants to define more complex combinations with discrete rules, one needs to utilise OR and AND combinations of the rule. The value of discrete rules can be of any string value, including a number saved as a string. The requirements for continuous rules differ from the ones defined for discrete rules. The allowed set of comparison operators for continuous rules includes the following ones: $= > >= < <=$ in. Additionally, the only valid values for this type of rule are numbers.

Based on the described requirements and domain knowledge, we designed a .json format for a simulation scenario file, presented as Listing 4.

```

{
  ...
  "prioritisation_rules": [
    {
      "priority_level": 1,
      "rules": [ [ {
        "attribute": "client_type",
        "comparison": "=",
        "value": "BUSINESS"
      } ] ]
    },
    {
      "priority_level": 2,
      "rules": [ [ {
        "attribute": "loan_amount",
        "comparison": "in",
        "value": [1500, 2000]
      } ] ]
    }
  ]
  ...
}

```

Listing 4. Representation of prioritisation rules in a simulation scenario file

We introduce prioritisation rules under `prioritisation_rules` section as an array of items. Each item consists of two fields: `priority_level` and `rules`. The former field describes a priority level assigned to the process instance when a specified rules is true. This priority level is of number type, and duplicates are not allowed. The later field, `rules`, represents a rule itself. We follow the same structure of OR and AND combinations inside the rule. The outer array represents OrRule entity, while the inner one - AndRule entity. The object contained within the inner AND array describes the smallest entity - Rule. It contains three fields: `attribute`, `comparison`, `value`. The first field, `attribute`, specifies a case attribute, one from a list defined by a user in the earlier section. The second field is a comparison operator. The only allowed operators are `=` for discrete case attributes and `in` for continuous case attributes. The last field involved in the rule formation is `value`, and its value depends on the type of a case attribute. For discrete case attributes, it should contain string, while for continuous ones - array with two numbers where the first value is a lower boundary of a range and the second one is a higher boundary. Consequently, we sum up that the only allowed comparison for continuous case attributes is a range comparison. When evaluating a rule, we evaluate whether a value of a case attribute is between a specified range, including boundaries. In addition, this format allows us to specify a comparison of equality by putting the same value as a lower and upper boundary.

UI Design In the scope of this feature, we introduce two new tabs to the web application. Figure 15 depicts both of them: subfigure (a) represents defining a new case attribute, while subfigure (b) - a new case-based priority level.

The section for defining case attributes, depicted in Figure 15a, contains two main buttons *New discrete case attribute* and *New continuous case attribute*. Both of them have a built-in template which simplifies the user experience. Once a user clicks on one of them, a new case attribute section is added on the UI with prefilled data for name and valueConfig based on the attribute's type. For discrete attributes, valueConfig includes one option *option name* with a probability of 1, meaning that this case attribute has *option name* as a value in all simulated case instances. For continuous values, the default distribution function norm is provided. Having those default values makes it easier for an end-user to interact with components to specify the desired setup. This section also includes validations: for providing the unique case attribute name and having a sum of probabilities equals 1 for continuous attributes.

The next section, illustrated in Figure 15b, is used to configure the priority levels based on the case attributes. *New prioritisation rule* button adds a new item to the list with prefilled data: priority level of 1 and enabled rule with the default template containing one AND rule. Every atomic rule contains three inputs: *field*, *operator*, *value*. *field* input contains options for an end-user: all case attributes that were defined in the previous tab *Case Attributes*. While selecting a discrete case attribute as a *field*, *value* input is transformed into a dropdown menu with a list of all possible values for the selected case attribute. In case of a continuous case attribute selected as a *field* in a rule, a user is eligible to put any number as a *value*.

Execution Semantics In order to perform the case-based prioritisation, we need to support a generation of case attributes.

So, first, we add this support to the console application. Generation of case attributes implies that every process case instance has an assigned case attribute value based on the user's setup. The assigned value does not change during the simulation process, so we calculate all case attributes' values at the beginning of the business process simulation. Algorithm 3 describes the logic behind this generation wrapped as *GenerateCaseAttr* function. As input, we receive two parameters: 1) a number of case instances generated in result of a process simulation, and 2) a list of setups per each case attribute. The former parameter dictates how many case instances we need to generate values, and the latter one describes how to generate those values. In line 5, *all_case_attr_dict* variable is assigned with an empty dictionary, which is a key-value storage. We store our final results in this variable. Then, we perform a loop the same number of times as we have process instances. Inside a loop, in line 7, we add a new item where *key* is an identifier of a process case at the current iteration and *value* is a list of case attributes for the case instance. This list is calculated by *GetValuesCalculated* function, presented in lines

Prosimos ONLINE BUSINESS PROCESS SIMULATOR

[← UPLOAD NEW MODEL](#)
[START SIMULATION](#)
[VIEW MODEL](#)
[DOWNLOAD AS A .JSON](#)

[Case Creation](#)
[Resource Calendars](#)
[Resources](#)
[Resource Allocation](#)
[Branching Probabilities](#)
[Intermediate Events](#)
[Batching](#)
[Case Attributes](#)
[Prioritisation](#)
[Simulation Results](#)

[+ NEW DISCRETE CASE ATTRIBUTE](#)
[+ NEW CONTINUOUS CASE ATTRIBUTE](#)

Case Attribute's Name
client_type

[DELETE](#)

Option List

Value	Probability	
REGULAR	0,8	+ NEW OPTION
BUSINESS	0,2	DELETE

Case Attribute's Name
loan_amount

[DELETE](#)

Value Distribution

Distribution Function	Scale (sec)	Min (sec)	Max (sec)
fix	240	0	1

(a)

Prosimos ONLINE BUSINESS PROCESS SIMULATOR

[← UPLOAD NEW MODEL](#)
[START SIMULATION](#)
[VIEW MODEL](#)
[DOWNLOAD AS A .JSON](#)

[Case Creation](#)
[Resource Calendars](#)
[Resources](#)
[Resource Allocation](#)
[Branching Probabilities](#)
[Intermediate Events](#)
[Batching](#)
[Case Attributes](#)
[Prioritisation](#)
[Simulation Results](#)

[+ NEW DISCRETE CASE ATTRIBUTE](#)
[+ NEW CONTINUOUS CASE ATTRIBUTE](#)
[+ NEW PRIORITISATION RULE](#)

Priority
1

[DELETE](#)

Condition

OR

AND

Field: client_type Operator: Equals Value: BUSINESS

Priority
2

[DELETE](#)

Condition

OR

(b)

Figure 15. UI sections, defining (a) case attributes, (b) multiple levels of case-based prioritisation

1-3. Here, we utilise an idea of list comprehension: creating a list based on another one. As each case attribute setup should result in a case attribute value, length of both those lists is equal. The function `GetNextValue` in line 2 generates the case attribute value based on the case attribute's setup. Setup includes such fields as a type of a case attribute and function distribution or list of possible values. Once we calculate values per each case attribute setup, we return this as a result. In case we have case attribute setup as provided in Listing 3, one of the possible results of `GetValuesCalculated` function is: `["REGULAR", "240"]`. After we calculate case attribute values per all process instances, we return the formed dictionary as a function result in line 9.

Algorithm 3: Generation of case attributes' values

```

1 function GetValuesCalculated (case_attributes: []):
2   | return [GetNextValue(attr) for attr in case_attributes];
3 end

4 function GenerateCaseAttr (total_num_cases: number; case_attributes: []):
5   | all_case_attr_dict  $\leftarrow$  dict();
6   | for case_id  $\leftarrow$  0 to total_num_cases do
7   |   | all_case_attr_dict[case_id]  $\leftarrow$  GetValuesCalculated (case_attributes);
8   | end
9   | return all_case_attr_dict
10 end

```

Once we have those values calculated, we are able to 1) use them for a case-based prioritisation; 2) output them as a part of a simulation log file as new columns. Each case attribute has its own column in a resulting file. The column's name corresponds to the name field provided by a user.

The next set of changes is made to support a case-based prioritisation. We already have case attributes and their values in place, so now can construct prioritisation rules based on case attributes following the `.json` structure.

The order of the rules specified in a simulation scenario is not playing any role. The system orders them based on the priority level (from the lowest priority to the highest, e.g., from 1 - to infinity). When calculating the priority level of an individual case, the check for the truthy rule is stopped once the first true one is found. This means that even when we have case attributes that satisfy two separate rules in the configuration, the one higher in the priority is taken as a final priority level.

Once a user defines priority rules and we parse them, we need to assign a priority level for each case instance. Algorithm 4 presents the calculation of the priority level of a case. The first input parameter is `all_case_values`, which lists all case attributes' names and their calculated values for a specific case instance. Another input parameter is `all_priority_rules`, containing information about all priority rules defined by a user in a simulation scenario. Line 2 states that, by default, a priority level is the maximum

possible integer number. This priority level means that a case is executed the last. If case attributes' values do not satisfy any of the rules, we return this maximum level of priority as a function result. We start with iterating over all defined rules in line 3. The next line defines the initial value of a rule outcome which is False. Since the outer array is an OR rule, we need at least one rule to be True to turn the rule outcome into True. Then, we start looping over OR rules in line 5. The initial value of AND rule outcome in line 6 is True. In order to conclude that this rule is True, we need to have all conditions to be equal to True. That is why the starting point should also be True. The next for-loop (line 7) is across an AND rule. At this stage, we already have the smallest possible atom which can be evaluated. For evaluation, we use `is_true` function, which compares an attributes' values of a specific case instance against a condition, `atomic_rule`. This function includes coverage of two cases inside: for both discrete and continuous case attributes. After we evaluate an atomic rule, we compare it with our previously received outcomes by using AND boolean operator in line 8. Once we finish evaluating all conditions inside AND rule, we return to evaluating an OR rule in line 10. Here, we compare the previous outcome with a new one with the help of OR boolean operator. Due to the fact that we need at least one True value in OR rule in order to evaluate a whole rule as True, line 12-15 describes a short circuit. As soon as we reach the first truthy value of one condition inside an OR rule, we conclude that this OR rule is True and return the priority assigned to this OR rule as a function result.

Algorithm 4: Get priority of a case instance

```

1 def GetPriority (self, all_case_values, all_priority_rules):
2     init_priority = sys.maxsize // by default, the lowest priority
3     for rule in all_priority_rules do
4         // evaluate a rule
5         or_res = False;
6         for or_rule in rule.or_rules do
7             and_res = True;
8             for atomic_rule in or_rule.and_rules do
9                 | and_res = and_res AND atomic_rule.IsTrue(all_case_values);
10            end
11            or_res = or_res OR and_res;
12        end
13        if or_res == True then
14            | init_priority = rule.priority;
15            | break;
16        end
17    end
18    return init_priority;
19 end

```

With priorities now assigned to each case instance, we introduce changes to the order of task execution. In *Prosimos 1.0*, we use a priority queue to decide which task is executed next and enabled datetime is a priority value for the queue. When introducing prioritisation, the priority queue remains to be used. However, the queue is extended and an updated version of the priority value contains a tuple of a case priority level and enabled time. When adding a task to or removing an item from the queue, we first take into account the priority level. In case an activity has the same priority level, we check the enabled time next and select the item (for execution or deleting) based on that. As a result, we take two parameters into consideration while prioritising the next activity for execution: 1) a priority level of a case under which an activity is being executed, 2) the enabled time of activity.

Additionally, when examining task execution, we must keep in mind the implications of both task prioritisation and batching. Previously, before introducing a prioritisation, we add a batch to a priority queue with an enabled time of a whole batch. However, now we also need a priority level of a batch. Consider the case when we have four activities in an enabled batch with the following calculated priority levels: 6, 2, 5, 4. We split this scenario into two parts, and the logic behind looks the following way:

- *Task execution outside a batch* describes how a batch is treated compared to other tasks in a queue. Both single tasks and batches co-exist in the same priority queue. This means that both of them require an enabled time and a priority level. An explanation of the priority levels for individual tasks is described in previous paragraphs. For batches, we establish a different approach: the highest priority of all tasks inside a batch is considered as a priority of the batch. Returning to our example, we calculate $\max(6, 2, 5, 4)$ and receive 6 as a final priority of the batch. Consequently, we add this batch to a priority queue with the priority level of 6.
- *Task execution inside a batch* specifies the order of task execution inside a batch itself. In this case, we adhere to the strategy used for a priority queue. This implies that tasks are executed in the order of a priority level assigned to a task's case. In our discussed example, the order is $order_ascendingly(6, 2, 5, 4) = 2, 4, 5, 6$.

5 Testing

Following the methodology of this thesis, we continue testing the developed artifact after its design and implementation. Testing is one of the core elements of the whole software development process, and, in some cases, 50% of project costs are spent for the testing part only [Ber07].

Our developed web application consists of multiple components, as presented in Figure 3. Since we want to allow users to either use a command line or web interface, we focus on the *Prosimos back-end* component as it is shared between those two interfaces. Hence, this section and the next one describe testing and evaluation of *Prosimos back-end* accordingly.

There are different approaches how to test the application during its development, including, but not limited to, unit testing, integration testing, system testing and end-to-end testing. In most cases, development teams use a combination of those. Our work concentrates on *decision table testing*. This technique is designed to test complex business rules. It belongs to a black-box testing method, meaning we do not test the statements (each line) inside functions but rather look at the software as a complete system. Consequently, we provide input to the system and expected output after performing one or many actions. We call the combination of individual input and its expected output a *test case*. We form those decision tables by combining different test cases together.

As *intermediate events* are a simple implementation, we cover this functionality with unit tests. On the contrary, *prioritisation* and *batching* introduce complex logic, so we designed tests with the help of decision tables. The following paragraphs present the design of those tables for each concept separately.

5.1 Case-based Prioritisation

One of the main parts of prioritisation is being able to calculate whether a priority rule is true. So we first design tables for those rules and start with atomic rules. Table 2 describes test cases for rules defined with a continuous case attribute. Rule evaluation requires the rule itself and the value of a case attribute. We split rule ranges into equivalence classes, meaning we observe the same behaviour of values inside those classes. We define three classes: 1) $[0, X]$ - a lower boundary is 0, represents $< or \leq$ operator, 2) $[X, Y]$ - both boundaries are natural numbers, except 0, 3) $[Y, \infty]$ - an upper boundary is an infinity (e.g., not defined in a rule), represents $> or \geq$ operator. Based on those rules, we define equivalence classes for the value of a case attribute, which can be inside or outside the range. As a result, the second condition takes only one of two values: either true or false. Additionally, if the range contains natural numbers as upper and lower boundaries, the value could be lower or greater than the range boundaries. Once we define possible options for all conditions of the rule, we write down the expected result in the table's last line, which is *Rule Evaluation* in our case.

Rule							
Condition	1	2	3	4	5	6	7
Rule Range	[0, X]	[0, X]	[X, Y]	[X, Y]	[X, Y]	[X, inf]	[X, inf]
Is Value in a Range?	True	False (greater)	True	False (lower)	False (greater)	True	False (lower)
Action	1	2	3	4	5	6	7
Rule Evaluation	True	False	True	False	False	True	False

Table 2. Decision table for evaluating a continuous atomic rule

Compared to continuous values, discrete values support only *equals* operations. Due to this, decision tables are overcomplicated solutions as there are few input values.

Having verified the logic of atomic rules, we move one level up and verify *AND* rules, which consist of atomic rules. For the decision table, we select AND rule containing two atomic rules. The structure of this rule looks as follows:

Simple Rule 1 AND Simple Rule 2

Table 3 depicts possible inputs and expected outcomes of the rule's evaluation. We have two conditions (inputs) for this AND rule, described by the atomic rule's outcomes. As a boolean expression represents all atomic rules, evaluation of the rules results in either a truthy or falsy value. Having two conditions and two possible outcomes for each condition, we end up with four rules in total. The last row in the table, *AND Rule Evaluation*, describes our expected result after a rule evaluation. Per the definition of AND rule, its evaluation outcome equals to true only in case all, in our case only two, atomic rules are true.

Rule				
Condition	1	2	3	4
Simple Rule 1 Outcome	True	False	True	False
Simple Rule 2 Outcome	True	True	False	False
Action	1	2	3	4
AND Rule Evaluation	True	False	False	False

Table 3. Decision table for evaluating AND priority rule

Based on the four rules mentioned above, we create test cases and verify an expected result. For example, for Rule 1, we specify two true atomic priority rules with the currently specified case attributes. Once we design them, we combine two rules to AND rule and run the rule evaluation. After that, we compare the evaluation outcome against the expected result. Test implementation can be reviewed via GitHub²⁵.

By combining AND rules, we form a new type of rule - *OR rules*. We design the decision table for this type of rule following the boolean algebra in the same way we do

²⁵https://github.com/AutomatedProcessImprovement/Prosimos/blob/main/testing_scripts/test_case_priority_is_true.py#L74

for AND rules. For this testing approach, we structure the OR rule in the following way by combining two complex (AND) rules:

Complex Rule 1 OR Simple Rule 2

As a result, the evaluation of the OR rule depends on the evaluation outcomes of those two parts: Complex Rule 1 and Complex Rule 2. Both of those rules are boolean with only two possible values: true or false. By making all possible combinations, we construct four rules, depicted in Table 4. Following the boolean algebra, we fill in expected values where evaluation outcomes are equal to true if at least one of the rules is true. Based on the decision table, we implemented a parameterised test which can be viewed in GitHub ²⁶.

Condition	Rule			
	1	2	3	4
Complex Rule 1 Outcome	True	False	True	False
Complex Rule 2 Outcome	True	True	False	False
Action	1	2	3	4
OR Rule Evaluation	True	True	True	False

Table 4. Decision table for evaluating OR priority rule

5.2 Batch Processing

Compared to prioritisation, batch activation rules are of a higher complexity due to the increased number of available types of attributes used for atomic rules. While we specify only two types of priority rules (discrete and continuous), batch activation rules include five type variations. Verifying all combinations of those is not a trivial task. So we also use decision table testing, which helps us to formulate all possible inputs and their results. We present a subset of decision tables to be able to explain the main approach.

Due to the relative simplicity of atomic batch firing rules, we omit designing decision tables for them. We covered those rules with unit tests instead. Consequently, we start with decision tables for the already complex rule. For batching, we design decision tables for a complete simulation run. This impacts our expected results not tied to a specific value or number but to behaviour in the resulting log file.

The first OR rule we test takes the following form:

(Daily Hour 1 AND Batch Size 1 AND Weekday 1)
OR
(Daily Hour 2 AND Batch Size 2 AND Weekday 2)

²⁶https://github.com/AutomatedProcessImprovement/Prosimos/blob/main/testing_scripts/test_case_priority_is_true.py#L102

The specified rule has six individual atomic rules, which values we can change. In order to reduce the complexity, we make weekdays for both parts of OR rule constant and equal to *Tuesday* and *Friday*. Table 5 presents values for the rest of the atomic rules present in the rule. Due to the time intensity of the simulation run, we aim to reduce the number of test cases to speed up the whole testing process. For this, we take the approach of combining changes to different conditions in one rule. For example, we change both *Daily Hour 1 & 2* and *Batch Size 1 & 2* in Rule 2, compared to Rule 1. This way, we reduce the number of rules but keep covering all equivalence classes. Regarding the action results, we check whether the resulting batches in the log file follow the size requirement (*Size of Batch Follows the Rule*) and follow the daily hour and weekday requirement (*Batch Timestamp Follows the Rule*). However, the assertion in our tests looks different and more complex than the one in the table. As we know the number of instances, starting timestamp and other details, we are able to predict the time and size of the batch execution. As a result, we leverage the knowledge passed as parameters to the simulation and assert against real timestamps and the number of items inside individual batches. The test implementation is available in the GitHub repository²⁷.

Condition	Rule		
	1	2	3
Daily Hour 1 & 2	[0, 12)	(12, 23]	(13,21)
Batch Size 1 & 2	[4, inf]	[0, 6]	[0, 6]
Weekday 1	Tuesday	Tuesday	Tuesday
Weekday 2	Friday	Friday	Friday
Action	1	2	3
Size of Batch Follows the Rule	True	True	True
Batch Timestamp Follows the Rule	True	True	True

Table 5. Decision table for simulation run with OR rule containing three attributes

Another example of a complex batching OR rule we discuss looks as follows:

(Time Since First AND Time Since Last)

Both atomic rules in that example belong to *waiting time* type of batch activation rule. Compared to other types of firing rules, waiting times have their distinction: it is impossible to define *infinity* as an upper boundary. This is done in order not to wait forever for the rule to turn true. So our equivalence classes might take either $[0, X]$ or $[X, Y]$, where X and Y are natural numbers, except 0. The former format represents $< or \leq$ situation, and the latter describes a value range.

Table 6 represents designed test cases for the rule. We follow the same strategy of mixing different combinations of equivalence classes. Additionally, this test does not run the whole simulation; we target a specific moment of time in a simulation. Due to this,

²⁷https://github.com/AutomatedProcessImprovement/Prosimos/blob/main/testing_scripts/test_batching_daily_hour.py#L233

we introduce *Is Batch Enabled by Rule?*. This value changes depending on the items waiting in the queue. In the test itself, we hardcode items in the queue, and by this, we set up this condition to a specific value required for a test case. Since we do not run the whole simulation, the expected values are changed. In this scenario, the action verifies whether there is a batch enabled for the execution, and if yes, a batch size and its start time are also calculated. Based on the action, we assert two values: batch enablement and whether the returned batch size and timestamp follow the waiting time from the rule. Test implementation is accessible through the GitHub repository²⁸.

Condition	Rule			
	1	2	3	4
Time Since First	[0, 1h]	[0, 1h]	(30min, 2h)	(30min, 2h)
Time Since Last	(30min, 2h)	(30min, 2h)	[0, 1h]	[0, 1h]
Is Batch Enabled by Rule?	False	True	True	False
Action	1	2	3	4
Is Batch Enabled?	False	True	True	False
Does Batch Follow the Rule Requirement?	N/A	True	True	N/A

Table 6. Decision table for simulation run with OR rule containing two waiting time attributes

The rest of the tests follow the same approach of splitting input parameters into equivalence classes, combining them and then evaluating the action. All tests implemented as a part of this work are placed under *testing_scripts* folder²⁹, and their filenames start with *test_* prefix.

5.3 Code Coverage

After writing tests, it is essential to measure how much of the code base is covered by tests. This helps identify parts of the code that was never executed during a test run. A *code coverage* is a common metric used to achieve this goal. In our work, we also calculate this metric to report on the code percentage covered by tests. There exist different types of code coverage metrics, such as statement coverage, branch coverage or path coverage. While statement coverage computes the percentage of lines (same as statements) executed during test running, branch coverage measures whether each branch's paths are executed at least once during a test run. In programming, a branch is a part of the code where a system takes a decision and, as a result, might follow an alternative path of the code.

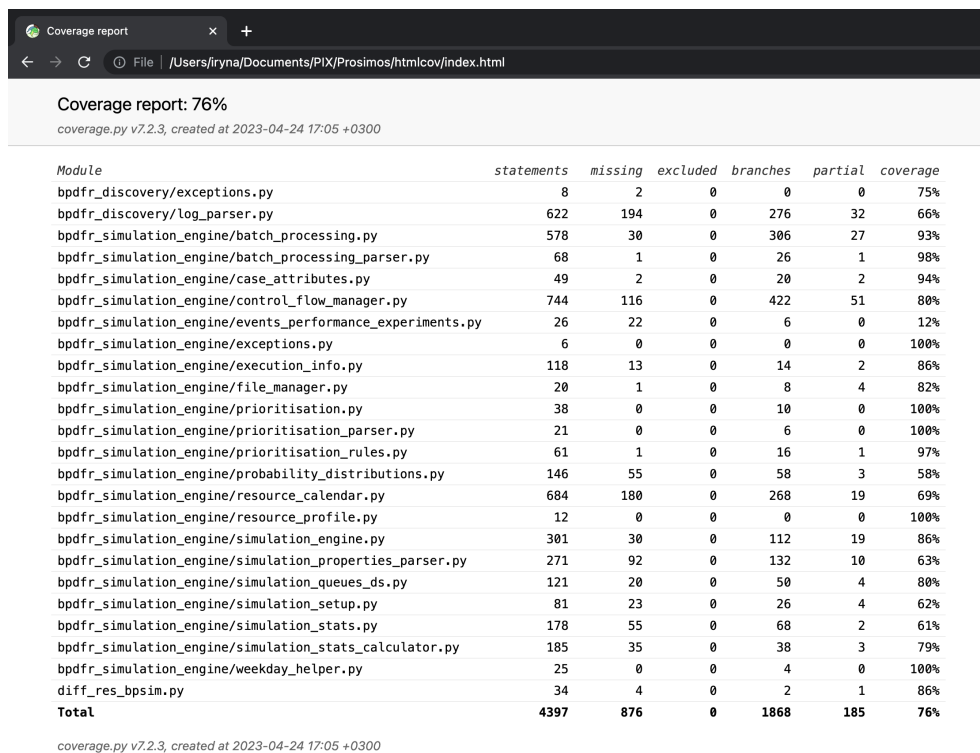
We use *branch coverage* metric to analyse our code coverage of *Prosimos back-end*. We choose this metric over a statement one because a branch coverage metric implies

²⁸https://github.com/AutomatedProcessImprovement/Prosimos/blob/main/testing_scripts/test_batching_large_wt.py#L347

²⁹https://github.com/AutomatedProcessImprovement/Prosimos/tree/main/testing_scripts

statement coverage. In other words, if branch coverage is 100%, this also means that statement coverage is 100%.

In regard to the testing framework, we use *pytest*³⁰. In addition to testing, this framework allows us to run a coverage report supporting different formats, including HTML, XML, terminal output and others. We use HTML format due to its comprehensiveness (colour highlighting of covered and not covered parts of the codes) and easiness of understanding (simple web page). Figure 16 depicts the branch coverage report over all functionalities implemented in *Prosimos back-end*. The total number of coverage is 76%. So we treat this number as follows: 76% of all branches are executed at least once during a test run. However, this number does not tell us anything about the correctness of the implemented functionalities concerning the user requirements. Decision tables are the solution which helps us with verifying the satisfaction of the requirements as they are a "bridge" between requirements and an expected result.



Coverage report: 76%

coverage.py v7.2.3, created at 2023-04-24 17:05 +0300

Module	statements	missing	excluded	branches	partial	coverage
bpdfr_discovery/exceptions.py	8	2	0	0	0	75%
bpdfr_discovery/log_parser.py	622	194	0	276	32	66%
bpdfr_simulation_engine/batch_processing.py	578	30	0	306	27	93%
bpdfr_simulation_engine/batch_processing_parser.py	68	1	0	26	1	98%
bpdfr_simulation_engine/case_attributes.py	49	2	0	20	2	94%
bpdfr_simulation_engine/control_flow_manager.py	744	116	0	422	51	80%
bpdfr_simulation_engine/events_performance_experiments.py	26	22	0	6	0	12%
bpdfr_simulation_engine/exceptions.py	6	0	0	0	0	100%
bpdfr_simulation_engine/execution_info.py	118	13	0	14	2	86%
bpdfr_simulation_engine/file_manager.py	20	1	0	8	4	82%
bpdfr_simulation_engine/prioritisation.py	38	0	0	10	0	100%
bpdfr_simulation_engine/prioritisation_parser.py	21	0	0	6	0	100%
bpdfr_simulation_engine/prioritisation_rules.py	61	1	0	16	1	97%
bpdfr_simulation_engine/probability_distributions.py	146	55	0	58	3	58%
bpdfr_simulation_engine/resource_calendar.py	684	180	0	268	19	69%
bpdfr_simulation_engine/resource_profile.py	12	0	0	0	0	100%
bpdfr_simulation_engine/simulation_engine.py	301	30	0	112	19	86%
bpdfr_simulation_engine/simulation_properties_parser.py	271	92	0	132	10	63%
bpdfr_simulation_engine/simulation_queues_ds.py	121	20	0	50	4	80%
bpdfr_simulation_engine/simulation_setup.py	81	23	0	26	4	62%
bpdfr_simulation_engine/simulation_stats.py	178	55	0	68	2	61%
bpdfr_simulation_engine/simulation_stats_calculator.py	185	35	0	38	3	79%
bpdfr_simulation_engine/weekday_helper.py	25	0	0	4	0	100%
diff_res_bpsim.py	34	4	0	2	1	86%
Total	4397	876	0	1868	185	76%

coverage.py v7.2.3, created at 2023-04-24 17:05 +0300

Figure 16. Branch coverage report of *Prosimos back-end*

Coverage report for the *Prosimos back-end* can be found under a *htmlcov.zip* archive in the repository³¹.

³⁰<https://docs.pytest.org/en/7.3.x/>

³¹<https://github.com/AutomatedProcessImprovement/Prosimos/blob/main/htmlcov.zip>

6 Evaluation

This section describes the empirical evaluation that was conducted after the completion of the tool development. We perform an experimental evaluation aiming to discover how our newly introduced concepts impact software performance.

As discussed earlier, we strive to evaluate *Prosimos back-end* component because it is shared between CLI and a web interface. Additionally, the execution of the simulation (*Prosimos back-end*) takes up most of the tool's performance if compared with the rendering part of the software (front-end). Consequently, these experiments are run using the console application only (*Prosimos back-end*).

We form three research questions in order to assess the tool's performance and scalability. In this evaluation, scalability determines how the software responds to an increasing number of input parameters, for example, a number of priority levels.

Three research questions are split accordingly to the introduced concepts. Since all types of intermediate events are treated similarly in the code base, there is no need to measure the performance of each type of intermediate event. So we pose a question generally without specifying the specific type. **RQ1:** *What effect does a number of intermediate events have on the simulation time?* While running experiments, we use *message* type of intermediate events. The next question will be split into two parts due to the complexity of the batch processing. We define two independent variables: *number of batched tasks* and *batching rule's complexity*. The first subquestion measures how the number of batched tasks impacts performance and is structured the following way: **RQ2-1:** *What effect does a number of batched tasks have on the simulation time?* Additionally, performance might be impacted by a number of levels of a firing rule. Hence, we formulate another question to assess the impact. **RQ2-2:** *What effect does the complexity of batching firing rules have on the simulation time?* The concluding concept we introduced is case-based prioritisation. Case-based prioritisation requires us to calculate the priority of each case only once, and this calculation happens at the beginning of the simulation. Hence, we avoid considering the independent variable *prioritisation rule's complexity* and form one research question. **RQ3:** *What effect does a number of priority levels have on the simulation time?*

Simulation time is a dependent variable in all presented research questions. This variable describes a simulation execution, starting from the parsing of BPS input (BPMN model and simulation scenario) and ending with a process execution and results saving to the files.

6.1 Datasets

We select one of the real-life logs used to assess *Prosimos 1.0* [LPHD23], namely log from the Business Process Intelligence Challenges (BPIC) of 2012³². The log describes a loan application process performed in one of the Dutch financial institutions. When selecting a dataset for evaluation, we focus on 1) model complexity regarding a number of activities; 2) presence of gateways and their types. We aim to choose a relatively complex model. BPIC-2012 model contains six activities, one inclusive (OR) gateway, a few exclusive (XOR) gateways, and 36 sequence flows. Based on this model description, the BPIC-2012 model appears to be a suitable match for evaluation runs. Additionally, the number of traces in the original log is 8 616, which results in an average process simulation time of 30 seconds. This time allows us to gauge the simulation time difference once we change the input parameters. A simulation time of one second makes it more challenging to assess the impact of input parameters due to insignificant duration change.

6.2 Experimental Setup

Experiments were run using the personal computer with 3.1 GHz Dual-Core Intel Core i5 processor, 16 GB of memory, and Intel Iris Plus Graphics 650 1536 MB graphics card.

In general, every simulation run returns different results. This happens because simulation is a stochastic process. As we strive to receive as accurate numbers as possible, we want to reduce the impact of randomness on our results. For this, we run five simulation iterations and then calculate a central tendency of simulation time over those iterations. As for the measure of central tendency, we select the median. We choose this metric over the mean to reduce the impact of outliers on the resulting number.

RQ1 In order to assess the impact of adding events to the process model, we need to have a mechanism on how to add them programmatically instead of doing it manually every time. For this, we implement a function which takes as input a BPMN model file and a number of events and returns a new BPMN model file with added intermediate message events per input specification. We insert events on every sequence flows between two elements. For example, if there are two consecutive activities in the model, we transform this part into a sequence of the following items: Activity \Rightarrow Intermediate event \Rightarrow Activity. The order of events insertion follows the order of sequence flow in a BPMN model file.

When running experiments, we start from no events in the process model till the maximum number of sequence flows, 36 in our case. Regarding specifying an event duration, this value remains constant throughout the whole experiment run and equals 15 min.

³²<https://doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f>

RQ2 Calculations required for the batch processing happen throughout the whole simulation run due to the nature of some types of firing rules. For example, waiting time rules are checked before and after the execution of the activity during the simulation run. These additional checks require more time and thus impact simulation time. Also, different types of firing rules have different logic on whether the rule is true and whether the batch is enabled. Consequently, we define here two independent variables: number of batched activities and complexity of the batching rule. The first independent variable defines how many activities are batched. We set up a range of values from 0 to the maximum number of activities in the model, 6 in our case. The complexity of the batching rule specifies how many levels a firing rule has. We define four levels of complexity:

1. R1
2. R3 AND R4
3. R1 OR (R3 AND R4)
4. (R1 AND R2) OR (R3 AND R4)

All types of batching rules have their own implementation and, as a result, different performance times when calculating whether the rule is true or batch is enabled. For experiments, we use all types of batching rules except the *ready_wt* due to its partial similarity with the *large_wt*. The rules appear as follows:

R1 *size* \geq 4

R2 *large_wt* < 3600

R3 *daily_hour* < 12

R4 *week_day* = Friday

The other fields required for the batch definition, such as *type* or *batch_frequency*, stay constant throughout the whole experiment.

We run a simulation for each level of complexity, changing the number of batched activities incrementally by one. As a result, we see how both variables, number of batched activities and level of complexity, impact the simulation time.

RQ3 In this part of the experiments, the only independent variable we have is number of priority levels. Per implementation, a user can specify any number higher than 0 as a priority level. We have a range of values from 0 to 4, where 0 refers to no priorities (meaning an empty array provided as a definition of prioritisation) and

4 - four priority levels. This range allows us to see the trend of the impact and derive conclusions. As we do not take into account complexity of prioritisation rules, all priority levels have only one simple rule over a continuous case attribute. A simple rule is one which contains neither AND nor OR part, e.g. `loan_amount in [1500, 2000]`.

6.3 Results

The explanation of the results is split into three parts based on the research question.

RQ1 Figure 17 depicts the correlation between the number of added events to the model and simulation time. The observed correlation is linear. We use Pearson's Correlation Coefficient (r)³³ to assess the strength of the correlation. After calculations, we received $r(34)=0.97$. The value in brackets (34) defines degrees of freedom and is calculated as $n - 2$, where n is a number of independent values. In our case, we have 36 independent values as we insert events for all 36 sequence flows. As a result $n = 36 - 2 = 34$.

Based on the coefficient value of 0.97, we conclude that the linear correlation is strongly positive: the more events we have in the model, the higher the simulation time is. An explanation for this lies in the implementation. Events use the same priority queue as all activities in the simulation engine. The time complexity of inserts to and removals from a queue is $O(\log n)$. As a result, the more elements we have in a queue, the longer it takes to add or remove an element from a queue.

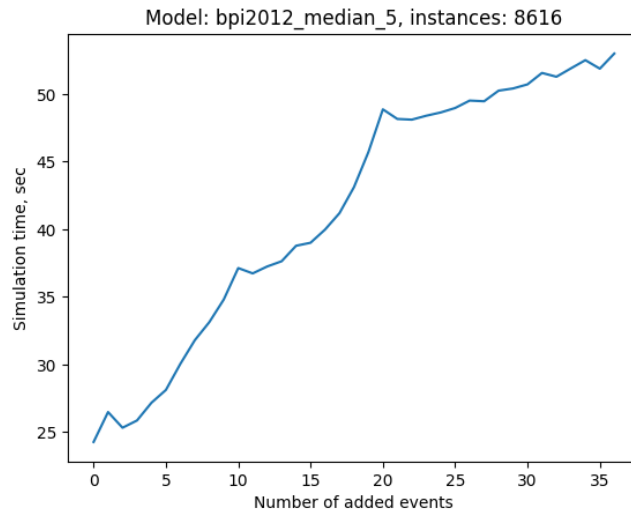


Figure 17. Correlation between the number of added events and simulation time (sec)

³³https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

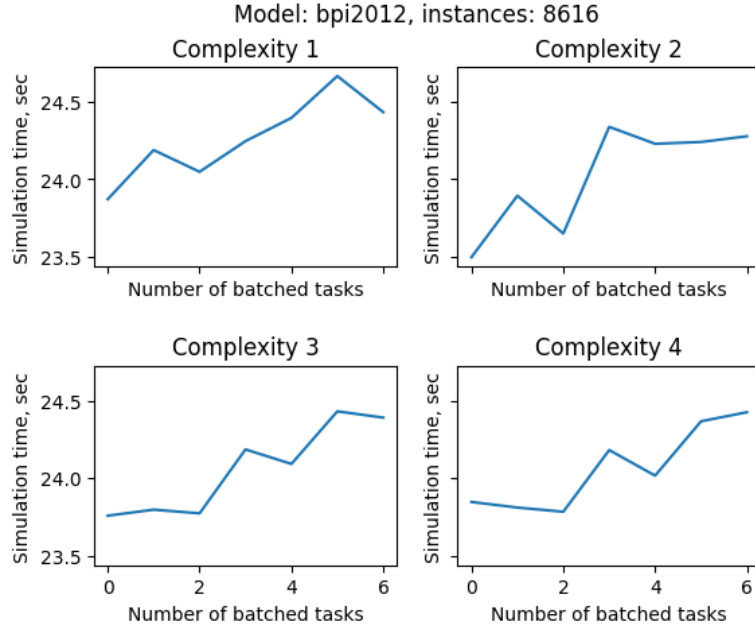


Figure 18. Relationship between the number of batched tasks and simulation time across four rule complexity levels

Furthermore, it is noticeable from Figure 17 that simulation time increases steeply in ranges of $[0-10]$ and $[10-20]$ events compared to the range of $[20-30]$ added events. It may be inferred that the simulation time of BPMN models with more than 20 events increases gradually. As we move from zero to 35 events in the BPMN model, we experience an approximately double increase in simulation time with a moderate rise after 20 events. We conclude that this trend is acceptable, and the feature scales well.

RQ2 Impact of a changing number of batched tasks and complexity level of batch-ing rules on the simulation time is presented in Figure 18. The plots show that the trend is the same for all levels of complexity.

Regarding the *number of batched tasks* (RQ2-1), the more batched tasks we have in the process model, the higher the simulation time is. Increasing number of batched tasks results in more computations of whether the task should be batched or whether the firing rule enables the batch to be executed. So received results behave the way we expect.

Regarding RQ2-2, Figure 18 does not expose any visible relationships between the complexity level and simulation time. For example, one can see that the simulation time differences between complexity levels three and four are almost invisible. Based on this, we conclude that the complexity level does not impact the simulation time. Accordingly, the software has good scalability when the rules contain four atomic rules combined as two pairs of OR rules. Yet users might define more sophisticated rules for batching, for example, containing ten pairs of OR rules or even more. In those types of cases, it is

possible that the system will scale differently. This assumption comes from the increased number of OR rules requiring additional evaluation of boolean expressions.

RQ3 Figure 19 shows how a number of prioritisation levels impacts the simulation time. It is hard to extract some dependency between those two variables as we can describe the relationship as neither linear nor exponential nor any other type of relationship.

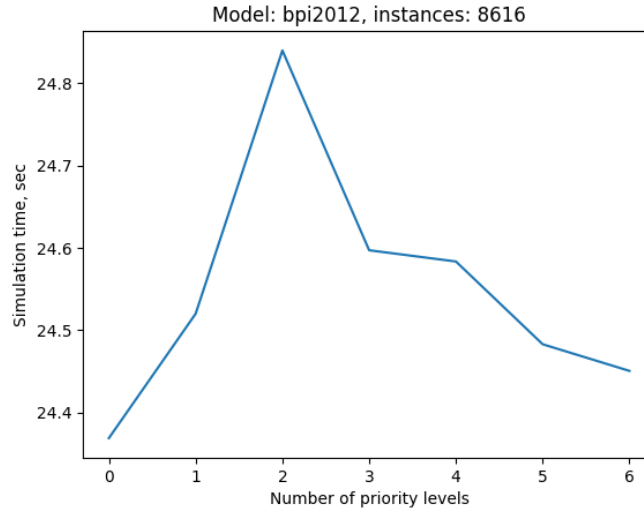


Figure 19. Relationship between a number of priority levels and simulation time (sec)

To better present the results, we create an additional plot where the y-axis shows the execution time of an individual activity. The dependent variable, execution time of an individual activity, is calculated by dividing the total simulation time by the number of executed activities. Figure 20 presents the received results. We observe a steep increase of 0.005 ms at the beginning when we move from no priority levels to one. After that, however, the simulation time of the individual activity starts to level off and is not drastically impacted by the number of priority levels.

Furthermore, we analyse the nature of the relationship in Figure 20 with more than one priority level. Table 7 presents execution times per activity per every priority level. We also calculate mean (M) and standard deviation (SD) values to gain data insights. Since our data do not contain outliers, we calculate an SD value instead of a mean deviation value. SD value of $8.54e-4$ shows that the execution times are dispersed quite closely to the mean, as a lower SD value implies that data points are closer to the mean.

Consequently, we infer that execution times per activity stay almost constant with a changing number of priority levels. We use the word *almost* because we identified the deviation in the execution times earlier. However, the obtained deviation is so small that

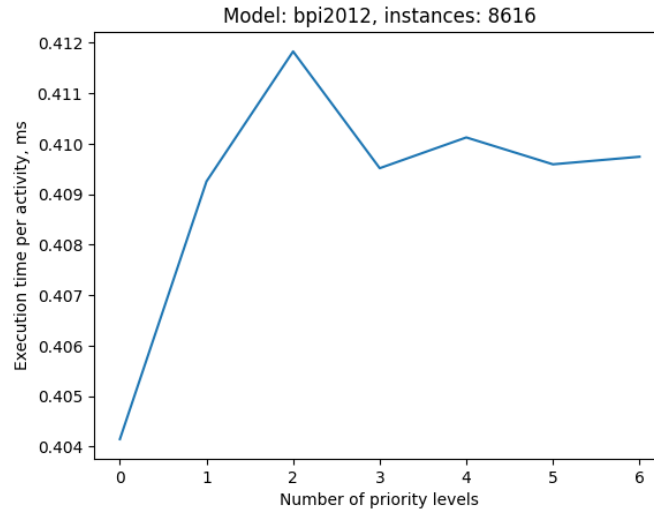


Figure 20. Relationship between a number of priority levels and executing time per activity (ms)

we can neglect it. This deviation happens due to some external events, such as other computer processes happening at the same time.

Number of priority levels	1	2	3	4	5	6	Mean (M)	Standard Deviation (SD)
Execution time per activity, ms	0.40925197	0.41182503	0.409513	0.41012231	0.40959003	0.40973995	0.41000705	8.54e-4

Table 7. All execution times per activity, their mean and standard deviation values

7 Conclusion and Future Work

This master's thesis was aimed at extending a business process simulation tool by 1) supporting additional elements at the control-flow level (intermediate events) and simulation scenario level (batch processing and task prioritisation) and 2) implementing a brand-new web application, including its deployment.

The first research goal was achieved by developing a new artifact of *Prosimos*. During the development process, unit and integration tests were written as a verification of the newly introduced concepts. Additionally, testing allows us to reduce maintenance costs in the future and have partial confidence that features introduced after do not impact the logic covered with tests. We achieved a branch coverage percentage of 76%.

Concerning the second goal, we performed an evaluation of the scalability possibilities per each concept. We run experiments measuring the simulation time. With the intermediate events, we experience a linear positive correlation between the number of added events and simulation time. While running experiments for batch processing, we considered two independent variables: the number of batched tasks and rule complexity. The simulation time linearly increases together with the increasing number of batched tasks. Regarding the rule complexity, it does not impact the simulation time. The last experiments investigated the impact of the number of priority levels. The received result was that simulation times stayed almost constant when changing the number of priority levels.

While we achieved our thesis goals, we might still enhance the simulation tool. Regarding the control-flow viewpoint, simulation lacks the support of subprocesses used to depict a complex business process. Another potential area of improvement might be to support other types of batching. Thus far, we implemented only two (parallel and sequential), but [LMCCD22] also defines other types. In this work, we extended the way tasks are executed by adding case-based prioritisation. However, there might be other behaviours which describe how prioritisation happens.

Furthermore, the implemented user interface has opportunities for improvement. Users might benefit from being able to save the simulation results and compare them with the following simulation run. Currently, comparing results from multiple runs can be done only manually after downloading the results of each run. In addition, visualising simulation results might benefit from making them more graphical instead of showing just tables. A combination of those two improvements (allowing users to compare multiple runs graphically) potentially improves the user experience.

Acknowledgments

This work is supported by the Estonian Ministry of Foreign Affairs – Development Cooperation and Humanitarian Aid funds.

References

- [Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, page 85–103, USA, 2007. IEEE Computer Society.
- [DBBMB16] Nadja Damij, Pavle Boškoski, Marko Bohanec, and Biljana Mileva-Boshkoska. Ranking of business process simulation software tools with dex/qq hierarchical decision model. *PloS one*, 11:e0148391, 02 2016.
- [DRMR18] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer Publishing Company, Incorporated, 2nd edition, 2018.
- [FP15] António Paulo Freitas and José Luís Pereira. Process simulation support in BPM tools: The case of BPMN. 2015.
- [Gro13] Object Management Group. Business Process Model and Notation (BPMN), Version 2.0.2. <http://www.omg.org/spec/BPMN/2.0.2/>, 2013.
- [HB94] James Higginson and James H. Bookbinder. Policy recommendations for a shipment-consolidation program. *Journal of Business Logistics*, 15(1), 1994.
- [HLD12] Zhengxing Huang, Xudong Lu, and Huilong Duan. Resource behavior measure and application in business process management. *Expert Systems with Applications*, 39(7):6458–6468, 2012.
- [HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.
- [LMCCD22] Katsiaryna Lashkevich, Fredrik Milani, David Chapela-Campa, and Marlon Dumas. Data-driven analysis of batch processing inefficiencies in business processes. In Renata Guizzardi, Jolita Ralyté, and Xavier Franch, editors, *Research Challenges in Information Science*, pages 231–247, Cham, 2022. Springer International Publishing.
- [LPD22] Orlenys López-Pintado and Marlon Dumas. Business process simulation with differentiated resources: Does it make a difference? In *Business Process Management*, pages 361–378, Cham, 2022. Springer International Publishing.

- [LPHD23] Orlenys López-Pintado, Iryna Halenok, and Marlon Dumas. Prosimos: Discovering and simulating business processes with differentiated resources. In *Enterprise Design, Operations, and Computing. EDOC 2022 Workshops: IDAMS, SoEA4EE, TEAR, EDOC Forum, Demonstrations Track and Doctoral Consortium, Bozen-Bolzano, Italy, October 4–7, 2022, Revised Selected Papers*, pages 346–352. Springer, 2023.
- [MS06] Mathirajan M and Appa Iyer Sivakumar. A literature review, classification and simple meta-analysis on scheduling of batch processors in semiconductor. the international journal of advanced manufacturing technology, 29, 990-1001. *International Journal of Advanced Manufacturing Technology*, 29:990–1001, 07 2006.
- [Mur23] Alexandra Murtaza. 8 Top Advantages of Using React for Development. <https://www.creative-tim.com/blog/educational-tech/top-advantages-of-using-react/>, 3 2023.
- [PW13] Luise Pufahl and Mathias Weske. Batch Activities in Process Modeling and Execution. pages 283–297, 12 2013.
- [RATE05] Nick Russell, Wil Aalst, Arthur Ter, and David Edmond. Workflow resource patterns: Identification, representation and tool support. volume 3520, pages 216–232, 06 2005.
- [RM05] H. Reijers and Selma Mansar. Best practices in business process redesign: An overview and qualitative evaluation of successful redesign heuristics. *Omega*, 33:283–306, 08 2005.
- [Sem23] Wojciech Semik. Flask vs. Django: Which Python Framework Is Better for Your Web Development? <https://www.stxnnext.com/blog/flask-vs-django-comparison/>, 1 2023.
- [SS20] K. Schwaber and J. Sutherland. *The Definitive Guide to Scrum: The Rules of the Game*. Scrum.org, 2020.
- [SWX⁺17] Suriadi Suriadi, Moe T. Wynn, Jingxin Xu, Wil M.P. van der Aalst, and Arthur H.M. ter Hofstede. Discovering work prioritisation patterns from event logs. *Decision Support Systems*, 100:77–92, 2017. Smart Business Process Management.
- [zMR08] Michael zur Muehlen and Jan Recker. How Much Language Is Enough? Theoretical and Practical Use of the Business Process Modeling Notation. 03 2008.

Appendix

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Iryna Halenok**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

Business Process Simulation with Differentiated Resources,
(title of thesis)

supervised by Orlenys López-Pintado and Marlon Dumas.
(supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Iryna Halenok
07/05/2023