Petra : Process model based extensible toolset for redesign and analysis

Petra: Process model based Extensible Toolset for Redesign and Analysis

D.M.M. Schunselaar*, H.M.W. Verbeek*, W.M.P. van der Aalst*, and H.A. Reijers*

Eindhoven University of Technology,
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
{d.m.m.schunselaar, h.m.w.verbeek, w.m.p.v.d.aalst, h.a.reijers}@tue.nl

Abstract. In different settings, it is of great value to be able to compare the performance of processes that aim to fulfill the same purpose but do so in different ways. *Petra* is a toolset for the analysis of so-called process families, which support the use of a multitude of analysis tools, including simulation. Through the use of *Petra*, organisations can make an educated decision about the exact configuration of their processes as to satisfy their exact requirements and performance objectives. The CoSeLoG project, in which we work together with 10 municipalities, provides exactly the setting for this type of functionality to come into play.

1 Introduction

Within the CoSeLoG project, we are cooperating with 10 different Dutch municipalities. Each of these municipalities has to legally provide essentially the same set of services to their citizens but each of these municipalities do in subtly different ways [1]. Such "colour locale" gives different solutions to the same problem. Interestingly, by interpolating between the solutions, a solution space with unseen and potentially better process models is spanned.

There are few approaches capable of analysing a predefined solution space to identify an attractive design for a particular business process. The ones existing are based on configurable process models, but they are tailored towards a single tool yielding a limited set of performance information. In Business Process Redesign (BPR), the solution space is defined based on best practises and is aimed at finding the best process model instead of analysing a spectrum of process models. The advantage is that solutions can be found not present in any of the models. The disadvantages are that solution present in the models do not need to be found and the best practises are usually based on the intuition of a human expert. Furthermore, BPR approaches found in the literature all display any of the following deficiencies; they are limited in the performance indicators they take into consideration, they are defined with a single tool for analysis in mind, or they are only existent on paper, i.e., lack concrete tool support.

* This research has been carried out as part of the Configurable Services for Local Governments (CoSeLoG) project (http://www.win.tue.nl/coselog/).

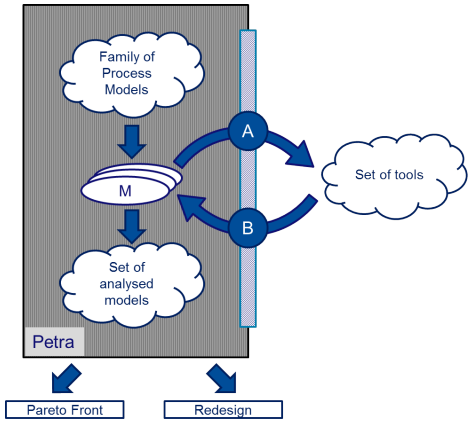

Fig. 1: Architecture of *Petra*. Given a set of process models, each of these is analysed using a set of tools and once analysed added to the set of analysed process models. Afterwards, the set of analysed process models can be reused, for example, to find the best model using a Pareto front or to support a process implementation effort.

We improve the state-of-the-art by providing an approach that is generic, extensible, and supported by automated tools. We provide a generic toolset called *Petra* (Process model based Extensible Toolset for Redesign and Analysis), see Fig. 1. In this toolset, there are 3 key concepts: (1) a family of process models (top left), (2) a (set of) properties (inside process models), and (3) a tool interface (bar with A and B). The process models are defined along 5 perspectives; control-flow, data, resource, environment, and experiment. *Control-flow*, *data*, and *resources* follow the classical subdivision of a process model along the perspectives. The *environment* perspective allows us to model behaviour out of control of an organisation, e.g., the arrival rate of new cases. The *experiment* perspective is necessary for controlling the experiments which determine whether a process model is indeed an improvement.

In order to store the results of the different tools, we have introduced the notion of a *property*. These properties can cover a range of things, e.g., expected costs, the expected number of handovers, or the sojourn time. Apart from using the properties for storing the results of the different tools, these properties also allow us to store run-time information making it possible to execute the model automatically. This information can vary from possible values a variable may take and thus the execution path a case takes, to specifying the work schedule of resources. Via the tool interface, we can use any analysis tool to enrich the process model with quality information. Furthermore, by not limiting the framework to a fixed set of properties, we are able to include any property of interest.

In the CoSeLoG project, we are working with configurable process models, in particular Process Trees. Configurable process models are process models which are enriched with configuration points. An organisation may want to use these to customise the configurable process model. For instance, an organisation may want to use the option to remove the payment process. Each configuration point has a set of configuration options denoting the possibilities at that point. By selecting for each configuration point an option, we obtain a process model. All the different process models obtainable by selecting different options for the configuration points forms the family of process models.

Apart from providing the architecture and implementation supporting our toolset, we also present a case study, which involves the application of our toolset on the models from the municipalities. This case study is twofold: (1) we show reality can be mimicked by our models, and (2) the municipalities can indeed learn from each other. The first part is shown by simulating the process model of the municipalities resulting in performance indicators highly similar to reality. The second part is shown by combining the models of the municipalities into a single configurable process model and by analysing the solution space spanned by this configurable process model.

The remainder of this paper is organised as follows: Section 2 presents the state-of-the-art as well as some tools for analysing process models. In Sect. 3, we present our high-level architecture in more detail. Building upon the high-level architecture, we present the parts within that architecture; the Process Trees are presented in Sect. 4, the properties currently used within our framework are stated in Sect. 5, the transformation used for our first tool (CPN Tools) is in Sect. 6, and the actual implementation of our toolset (Sect. 7). After presenting the key concepts of our framework, we show with a case study the applicability of *Petra* on a process from the municipalities (Sect. 8). Finally, in Sect. 9, we present our conclusions and sketch a panorama of future extensions and research directions.

2 Related Work

Our research touches on a number of topics. First, from our input, we have *configurable process models*. Second, to be able to store and compute KPIs, we use *properties* and *analysis techniques* on our process models. Finally, we can use *Petra* for *business process redesign* of process models. Each of these is treated in isolation in the remainder of this section.

2.1 (Configurable) Process Models

Configurable process models have been defined for different modelling formalisms, e.g., configurable EPC [2,3], configurable YAWL [4], configurable BPEL [4], and CoSeNets [1]. The first 3 formalism are more expressive than the latter, i.e., the CoSeNets are based on block-structured process models. However, with the first 3, explicit attention has to be paid to the soundness [1] of the configured model.

Therefore, we select CoSeNets and extend these since CoSeNets only focus on the control-flow perspective.

The approach that is most closely related to ours is presented in [5]. The configurable process models used in this approach are modelled in Protos. By converting the configurable process model into a CPN model, the authors can use the same model to analyse multiple variants of the same process model. The main limitation of their approach is the fact that they focus on a single tool (CPN Tools) which results in a non-extensible set of analysis results. Furthermore, their resource model is rather simplistic, i.e., all resources are available all of the time, no support for data, i.e., exclusive choices are chosen randomly, and for soundness, state space analysis has to be employed (see [6], for an overview on the importance of correctly modelling resources).

In [7], a questionnaire-based approach is presented to deduce on a high level the requirements a user poses on the best instantiation of a configurable process model. This, in combination with functional requirements, results in an instantiated process model closest to the user's requirements. This approach does not give any performance information but it can be used to beforehand limit the to-be-analysed solution space.

The Provop framework contains context-based configuration of the configurable process model [8]. Within the Provop framework, there are so-called context rules which limit the configuration possibilities by including contextual information. These context rules specify, for instance, that if the user requests high quality, then certain other activities have to be included in the process model. As with the approach by [7], the focus of this approach is not on performance information but can again be used to limit the to-be-analysed solution space.

2.2 Performance Indicators and Analysis Techniques

Efforts have been made in enriching BPMN models with simulation information by the *WfMC* standard *BPSim* [9]. However, we allow, amongst others, for a richer resource model by supporting arousal-based working speeds. Furthermore, *BPSim* is tailored towards simulation and thus abstracting from non-simulation related information. We would like to be able to encode this non-simulation related information as this might be of importance for different analysis techniques. The properties are related to and inspired by *BPSim* and efforts are made for a transformation from properties to *BPSim* to be able to work with a standardised interface towards simulation tools.

Analysis techniques can be subdivided into two groups: quantitative analysis and qualitative analysis. Quantitative analysis methods are mainly related to, amongst others, computational techniques. Qualitative analysis methods are mainly related to expert views on the quality. We focus mainly on quantitative analyses, i.e., we persue automated techniques for analysing our process models.

Quantitative analysis methods focus on a broad range of qualities of a process model. Some focus on correctness, e.g., Woflan [10], while others focus on performance related information, e.g., queueing theory [11], simulation [6], and flow analysis [12].

As our first focus is on simulation, we list some of the tools: BIMP[1], CPN-Tools [13], GreatSPN [14], HPSim[2], Ina[3], JMT [15], Petruchio[4], PiPe2 [16], PNetLab[5], TimeNet [17], and Tina [18]. From these, we have selected CPN Tools as our first tool for 3 reasons: (1) Flexibility in defining the to-be-computed characteristics, (2) by using Coloured Petri Nets, we can easily encode the more advanced insights in resource behaviour, and (3) through Access/CPN [19], it is possible to simulate all the models without human involvement.

2.3 Process improvements and redesign

Within the field of BPR, different frameworks exist with quality metrics of a process, e.g., the Devil's Quadrangle [20]. In [21], more relevant related work with respect to BPR is provided. However, most of these approaches reason on a more conceptual level, and are not applicable to our problem setting. Therefore, we limit ourselves to operational approaches, i.e., approaches which, if implemented, can be applied directly onto the process models under consideration.

An approach related to our approach is presented in [22]. On the existing process model, process measures are computed. Afterwards, different applicable best practices are evaluated and applied to obtain new process models. Finally, on these process models, process measures are computed, the best process models are kept, and the steps are repeated. In [23], an implementation is presented. However, the focus is on simulation and time-related information. Furthermore, a single tool is selected for the analysis, making it impossible to obtain information not provided by this specific tool.

Different approaches exist which generate from a model different variants using a library of fragments, e.g., [24, 25]. The main idea of these approaches is: given a process model, substitute different part of the given process model with fragments from the library. Afterwards, evaluate the created process models and possibly apply the previous steps again. However, the main question remains how this library of fragments is to be constructed. Furthermore, the approaches exist on paper and tool support is lacking.

In [26], an approach is presented using process mining and simulation. First, the control flow of a process model is mined as a heuristic net. Afterwards, the control flow is transformed to a CPN model and timing information is added. Finally, the CPN model is simulated, and possible improvements are to be manually established. The resource perspective is overly simplistic, i.e., the only moment resources are mentioned is during the improvement phase where a manual activity is replaced by an automated activity. Finally, finding improvements is a manual step.

[1] http://bimp.cs.ut.ee/, last accessed 2013-17-12

[2] http://www.winpesim.de/hpetrisim.html, last accessed 2013-17-12

[3] http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/ina/, last accessed 2013-17-12

[4] http://petruchio.informatik.uni-oldenburg.de/, last accessed 2013-17-12

[5] http://www.automatica.unisa.it/PNetLab%20Download%20Page/pnetlab.htm, last accessed 2013-17-12

More specific approaches seek to find optimisation by modifying the structure of the process model. For instance, in [27–29] the authors propose to maximise the set of activities which are executed in parallel as to minimise the sojourn time. In [30], different metrics are defined on the structure. Making modifications on the syntax of the model disregards the dynamic behaviour, e.g., putting parts of the model in parallel might be counter productive. For instance, if one puts activities in parallel which are only executable by a single resource, there will not likely be an improvement whilst the wait time increases.

Finally, the above approaches mainly focus on steady-state behaviour. In [31], an approach is presented for short-term simulations. The authors in question use YAWL [32] as a workflow management system; from the current state they generate a simulation model. By simulating the model, they seek short-term improvements to cope with unexpected situations, e.g., in the paper they mention the situation where suddenly there is a large increase in arriving cases. This approach can be extended for steady-state analysis, i.e., for steady-state analysis the initial state does not matter and one needs to simulate for a longer period of time. However, the approach is tailored towards a single process model where one tries to optimise, for instance, resource allocation. Furthermore, they are focussed on a single organisation, i.e., all the redesign and improvements have to come from the organisation itself and there is no mechanism to learn from others.

The approaches under consideration, have either of following weaknesses: (1) it is generic but not implemented, (2) it is implemented but focusses on a predetermined set of types of qualities, e.g., syntactical qualities, or sojourn time, (3) no systematic analysis of a predetermined set of process models, or (4) overly simplified perspectives, in particular the resource perspective. Making none of the approaches applicable for embarking on our quest to find the best process model.

There are some approaches close to our approach, e.g., [5], [22], and [31], but these all use a single tool focussed on a limited set of performance indicators with oversimplified data and resource perspectives. We improve the state-of-the art by being generic, extensible with new tools and performance indicators, and allowing for a richer encoding of data and resources.

3 Architecture

To analyse the solution space spanned by a configurable process model, we propose *Petra*, the toolset depicted in Fig. 1. In *Petra*, we have a family of process models from which we want to select the best-fitting model, i.e., the model best for the set of requirements of the organisation at a particular moment in time. We extend every process model with relevant properties computed using different tools, e.g., sojourn time, queue time, costs, etc. At any moment in time, a tool can indicate that further computations on a particular process model are futile, e.g., the process model does not adhere to legal requirements or the utilisation of some resources is above 1. After obtaining the relevant properties, the

process models are added to a set of process models. At a later stage, the user can, via different views on this set of models, obtain the desired process model.

As mentioned, *Petra* consists of 3 key concepts: family of process models, set of properties, and a tool interface. In the remainder of this section, we discuss these different parts.

3.1 Family of process models

Process Trees consist of 5 perspectives: control flow, data, resources, environment, and experiment. The *control flow* denotes the causal relationship between the activities which contribute to the completion of a case. The *data* perspective is related to the variables and guards used in the process model. The *resources* are related to who may execute which activity. The uncontrollable behaviour is encapsulated in the *environment* perspective, i.e., everything the organisation does not have any control over. Finally, the settings for an experiment are encoded in the *experiment* perspective, e.g., the parameters for the simulation.

Instead of having every process model specified separately, the family of process models are encoded in a Process Tree. By using a configurable process model, we remove any ambiguity which might arise when comparing two individual models. For example, if the configured yet different process models contain an activity with the same name, it is ensured that they actually refer to the same activity. In this configurable process model, all perspectives are configurable, e.g., we have the option to remove activities, change allocatability of resources, change which variables are used by a task. The Process Trees are described in Sect. 4.

3.2 Properties

For determining the best-fitting model, we need more information than just the control-flow, data, resource, environment, and experiment perspectives. Therefore, we introduce a generic notion of a *property*. Properties can be independent (facts), and dependent (deduced from facts). The set of properties is extendable and part of the current set of properties is inspired by the properties as defined in the *WfMC* standard *BPSim* [9]. By not limiting the framework to a fixed set of properties, we maintain the genericity of our framework. Properties can be part of every construct, e.g., resources, data, guard, edges, nodes, the process model itself, etc.

The independent properties are properties defined by the user before any analysis has been performed, e.g., the intensity of the arrival of cases and the different values a variable may take. Dependent properties are properties returned by the tools, e.g., sojourn time and costs. Note that a tool may require both dependent and independent properties, i.e., before being able to use a particular tool, another tool has to be used to compute certain properties.

Properties need to support a number of operations. The foremost operation is to define a comparator. This comparator allows for reasoning over the different instantiations of a property with a concrete value to find the best value (if

an ordering is possible). Furthermore, properties need to define a default value (identity element). This default value can be used in case a tool cannot handle empty values. For instance, if there is no specification of the values a variable may take, we need a default value because otherwise the expressions on the outgoing edges of choice constructs cannot be evaluated.

The properties and more requirements on those properties are described in more detail in Sect. 5.

3.3 Tool interface

To enrich the Process Trees with properties, tools can be used. If a tool does not immediately work on Process Trees, a wrapper has to be provided to transform the Process Tree to the tool-specific formalism (interface A). After the tool has finished, it should enrich the Process Tree from interface A with the newly obtained results and return this enriched Process Tree on interface B.

The properties offered on interface A can both be independent (defined by the user), and dependent (deduced by other tools). In our framework, we assume an incremental approach, i.e., tools may only modify/overwrite or add dependent properties, and are not allowed to remove (in)dependent properties.

In order to know which tools to use for the different properties, we require that a tool registers which (in)dependent properties it requires. Furthermore, to allow for chaining of tools for obtaining certain properties, we require a tool to register which dependent properties it computes. Finally, to prevent further computations due to issues detected by a tool, tools may signal that future processing of a process model by other tools is useless, e.g., there was no steady state in the simulation. We have chosen to encode this on the interface level instead of a property. The reason for this is that the toolset is oblivious to the properties used and it directly impacts the behaviour of the toolset.

Since the amount of possible models can be exponential in the amount of configuration points, we require a tool to run automatically, i.e., without the need of human involvement. Therefore, for the first tool to be used in the toolset, we have chosen CPN Tools because of the existence of Access/CPN [19]. Access/CPN provides an interface to run the simulator without any human involvement. The transformation of a Process Tree to a CPN model is explained in Sect. 6.

Having the architecture of our framework in place, we first introduce our process model formalism: Process Trees. Afterwards, we present the properties as well as an implementation to use and compute these with a concrete tool, CPN Tools [13], in our framework. Finally, in Sect. 7 the implementation of *Petra* is presented. Note that, the ideas behind *Petra* transcend the exact formalism used for modelling process models.

4 Process Trees

Process Trees are block-structured process models, a subclass of Free-Choice Petri nets, where every block has a single entry and a single exit. Process Trees

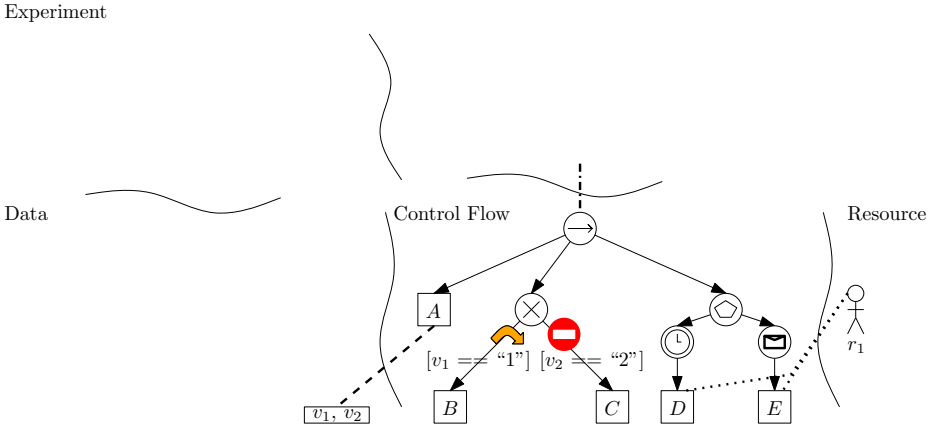

Fig. 2: The different perspectives in a Process Tree. The experiment and environment perspectives only consist of properties. The data perspective contains the variables and expressions and their relation to the control-flow perspective. The control-flow perspective specifies the execution sequence of activities. Finally, the resource perspective specifies which resources may execute which activity.

are related to the RPST [33] in the sense that both work on a block-structure. However, the RPST is used to convert non-block structured models into block structured models and focusses on the control-flow perspective.

A Process Tree consists of 5 perspectives: control-flow, resource, data, environment, and experiment. In Fig. 2, the different perspectives are depicted and their relation to each other. Each of the perspectives contains a set of configuration points with configuration options. In the remainder of this section, we go into each of the perspectives and show the configuration options.

4.1 Control-flow perspective

The control-flow perspective consists of nodes, and directed edges between these nodes which denotes the order of execution. Nodes come in two different flavours: tasks and blocks. Tasks are the units of work which need to be executed, e.g., A in Fig. 2, and can be manual (with a resource) or automatic. Blocks specify the causal relationship between its children, e.g., the block with the arrow in Fig. 2 denotes a sequential execution. All the nodes are depicted in Fig. 3 with their Petri net semantics[6]. Note that, we use some of the notations for events in Petri nets from [34].

If there is an edge from a block to a node, then we say that this node is a child of that block. For all blocks except for the AND, DEF, and EVENT the order of the children matters, therefore, there is a total order on the outgoing edges

[6] Note that, for many nodes, Fig. 3 shows the semantics for the case with only two children. However, it is trivial to extend this to more children.

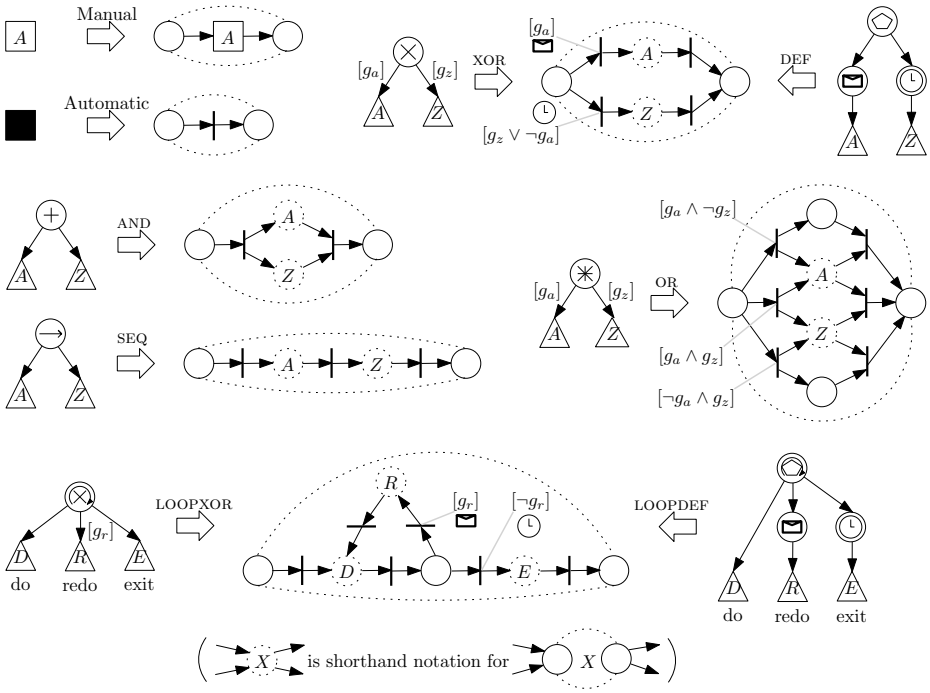

Fig. 3: The different nodes and their (abstract) Petri net semantics.

of a block. In general, all nodes can have any number of children excepts for the EVENT, LOOPXOR, and LOOPDEF (see Fig. 3). Finally, the set of nodes and edges forms a connected Directed Acyclic Graph (DAG) with a single root node, i.e., a node without any incoming edges. This DAG is a more compact representation of the tree since equivalent parts can be placed on top of each other. This is for the sake of maintainability and does not influence the expressive power of the Process Tree. Note that, for most tools, this DAG will be unfolded to a tree structure as the equivalence between the parts does not need to hold after computing different characteristics. For instance, the sojourn time for the same subprocess can be different dependent on where the subprocess is located in the process model.

The formalisation of the control-flow perspective P_f is:

Definition 1 (Control-flow perspective). *A control flow perspective is a 4-tuple* $(N, E, r, order)$ *where:*

- N *is the set of nodes, which is partitioned into the following sets:*
  - N_T *is the set of task nodes, which is partitioned into the following sets:*
    - N_A *is the set of automatic tasks*
    - N_M *is the set of manual tasks*
  - N_B *is the set of block nodes, which is partitioned into the following sets:*

        * N_{SEQ} *is the set of* SEQ *nodes*
        * N_{XOR} *is the set of* XOR *nodes*
        * N_{AND} *is the set of* AND *nodes*
        * N_{OR} *is the set of* OR *nodes*
        * N_{DEF} *is the set of* DEF *nodes*
        * N_{LOOPXOR} *is the set of* LOOPXOR *nodes*
        * N_{LOOPDEF} *is the set of* LOOPDEF *nodes*
        * N_{EVENT} *is the set of* EVENT *nodes*
- $E = \{(n, m) \mid m \in order(n)\}$ *is the set of edges such that the graph* (N, E) *is a directed acyclic graph. If* $(n, m) \in E$ *then* n *is called a parent of* m *and* m *is called a child of* n*.*
- $r \in N$ *is the unique node from which all other nodes in the graph* (N, E) *are reachable, that is,* $\{n \in N \mid (r, n) \in E^*\} = N$*, where* E^* *is the transitive and reflexive closure of* E*.*
- $order \in N_B \to N^+$ *imposes an order on the child nodes of a block. This is required for, for example, a sequence node.*

There are three types of configuration options: hiding, blocking, and substitution. Substitution entails replacing a part of the process model with a fragment from a predetermined collection of fragments (see [1]). Hiding, which is shown in Fig. 4 with the curved arrow, entails substituting a part of the process model with an automatic task, e.g., Fig. 4(b). Blocking, shown with a no-entry sign in Fig. 4, denotes the prevention of entering a part of the process model, e.g., Fig. 4(a). Note that blocking has non-local effects, e.g., if a part of a sequence is blocked, then the entire sequence becomes blocked. More formally:

Definition 2 (Configurable control-flow perspective). *A configurable control-flow perspective is a 6-tuple* $(N, E, blockable, hideable, r, order)$ *where:*

- N *is the set of nodes, which is partitioned into the following sets:*
  - N_T *is the set of task nodes, which is partitioned into the following sets:*
    * N_A *is the set of automatic tasks*
    * N_M *is the set of manual tasks*
  - N_B *is the set of block nodes, which is partitioned into the following sets:*
    * N_{SEQ} *is the set of* SEQ *nodes*
    * N_{XOR} *is the set of* XOR *nodes*
    * N_{AND} *is the set of* AND *nodes*
    * N_{OR} *is the set of* OR *nodes*
    * N_{DEF} *is the set of* DEF *nodes*
    * N_{LOOPXOR} *is the set of* LOOPXOR *nodes*
    * N_{LOOPDEF} *is the set of* LOOPDEF *nodes*
    * N_{EVENT} *is the set of* EVENT *nodes*
    * N_P *is the set of Placeholder nodes*
- $E = \{(n, m) \mid m \in order(n)\}$ *is the set of edges such that the graph* (N, E) *is a directed acyclic graph. If* $(n, m) \in E$ *then* n *is called a parent of* m *and* m *is called a child of* n*.*
- $blockable \subseteq E$ *is the set of blockable edges*

- *hideable* $\subseteq E$ *is the set of hideable edges*
- $r \in N$ *is the unique node from which all other nodes in the graph* (N, E) *are reachable, that is,* $\{n \in N \mid (r, n) \in E^*\} = N$*, where* E^* *is the transitive and reflexive closure of* E*.*
- $order \in N_B \to N^+$ *imposes an order on the child nodes of a block. This is required for, for example, a sequence node.*

Configuration A configuration for a configurable control-flow perspective selects for configuration points (i.e., placeholder nodes, and hideable/blockable edges) which option is taken, i.e., whether it is hidden/ not hidden, blocked/ not blocked, and which substitution option is selected/ are not selected. If for every configuration point a selecting is made, this configuration is total, else it is partial. A configurable control-flow perspective can be instantiated with a configuration to obtain a (partly) configured (configurable) control-flow perspective. More formally:

Definition 3 (Control-flow configuration). *Let* $(N, E, blockable, hideable, r, order)$ *be a configurable control-flow perspective, then a control-flow configuration* C_f *is a 6-tuple* $\left(N_b, \overline{N_b}, N_h, \overline{N_h}, r_{sel}, \overline{r_{sel}}\right)$ *where:*

- $N_b \subseteq blockable$*, the set of blocked edges*
- $\overline{N_b} \subseteq blockable$*, the set of non-blocked edges*
- $N_h \subseteq hideable$*, the set of hidden edges*
- $\overline{N_h} \subseteq hideable$*, the set of non-hidden edges*
- $r_{sel} \subseteq N_P \times N$*, the set of selected replacements for replacement nodes*
- $\overline{r_{sel}} \subseteq N_P \times N$*, the set of non-selected replacements for replacement nodes*

Definition 4 (Valid control-flow configuration). *Let* $(N, E, blockable, hideable, r, order)$ *be a configurable control-flow perspective, and let* $C_f = \left(N_b, \overline{N_b}, N_h, \overline{N_h}, r_{sel}, \overline{r_{sel}}\right)$ *be a control-flow configuration, then* C_f *is valid if and only if:*

- $N_b \cap \overline{N_b} = \emptyset$*, an edge cannot both be blocked and not blocked*
- $N_h \cap \overline{N_h} = \emptyset$*, an edge cannot both be hidden and not hidden*
- $N_b \cap N_h = \emptyset$*, an edge cannot both be blocked and hidden*
- $r_{sel} \cap \overline{r_{sel}} = \emptyset$*, a replacement option cannot both be selected and not selected*
- $\forall_{(p,n),(p',n') \in r_{sel}} (p = p' \Rightarrow n = n')$*, a placeholder node can have at most one replacement node selected*
- $\forall_{p \in N_P} (\{(p, n) \mid n \in order(p)\} \setminus \overline{r_{sel}} \neq \emptyset)$*, we cannot remove all replacement options from a placeholder node*

Note that, by transitivity an edge can both be blocked and hidden. In this case, we give precedence to hidden due to the non-local semantics of blocking, i.e., a hidden edge can become blocked by transitivity but a blocked edge cannot become hidden by transitivity.

Definition 5 (Total control-flow configuration). *A configuration* $C_f = \left(N_b, \overline{N_b}, N_h, \overline{N_h}, r_{sel}, \overline{r_{sel}}\right)$ *is total for a configurable control-flow perspective* $P_{cf} = (N, E, blockable, hideable, r, order)$ *if and only if:*

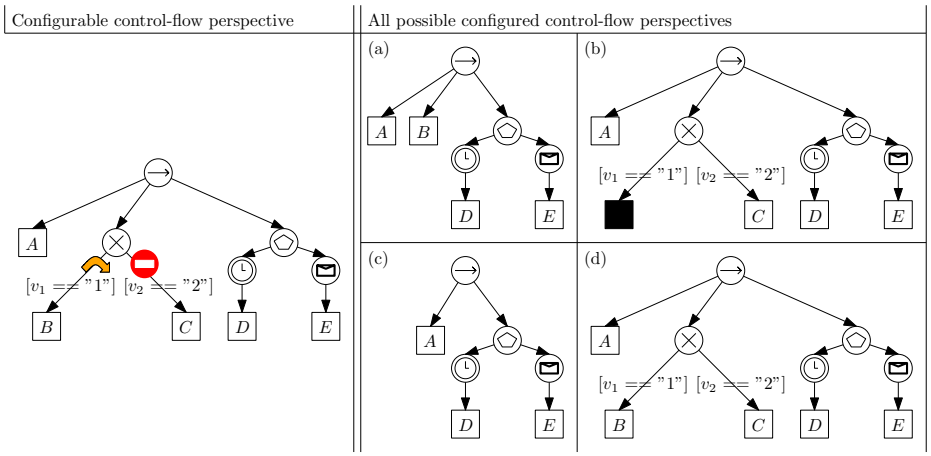

Fig. 4: The family of control flows spanned by the control-flow perspective using blocking and hiding.

- $(N_b \cup \overline{N_b}) = blockable$
- $(N_h \cup \overline{N_h}) = hideable$
- $(r_{sel} \cup \overline{r_{sel}}) = \bigcup_{p \in N_P} (\{(p, n) \mid n \in order(p)\})$

In Fig. 4, the space of models is depicted ((a)-(d)) using hiding and blocking. Figure 4(a) and (c), shows the configured model if we select blocking, Fig. 4(b) and (c) shows the configured model if hiding is selected and finally, Fig. 4(d) shows the configured model when none of the configuration options is selected. Note that, all models are obtained using total and valid configurations.

4.2 Data perspective

The data perspective specifies which expressions are used for the outgoing edges of a choice (between "[" and "]") and which variables are read and written by nodes (line from A to v_1 and to v_2 in Fig. 2). We have allowed to annotate all nodes with which variables are read or written. This allows for a subprocess to specify which variables are used in the subprocess. Furthermore, the XOR block requires variables in the evaluation of its expressions.

Expressions reason over variables and values for those variables and have the following operators; conjunction (&&), disjunction (||), and negation (!). Furthermore, variables may be compared to values on equality and inequality. More formally:

Definition 6 (Expression). *Let V be the universe of variables, let Val be the universe of values. An expression can be constructed using the following rules:*

- *If $v \in V$ and $val \in Val$, then* `v == val`*, and* `v != val` *are expressions,*
- *If p, q are expressions, then* `(p && q)`*,* `(p || q)`*, and* `!(p)` *are expressions.*

Definition 7 (Data perspective). *Let* $P_{cf} = (N, E, blockable, hideable, r, order)$ *be a configurable control-flow perspective, then a data perspective is a 5-tuple* $(V, Ex, G, read, write)$ *where:*

- V *is the set of variables*
- $Ex \subseteq \mathcal{V} \cup \{NoExpression\}$ *is the set of expressions, where* $\mathcal{V}$ *is the universe of logical expressions over* V*, and* $\{NoExpression\}$ *denotes that no expression is specified*
- $G \in E \to Ex$ *annotates edges with expressions guarding the execution*
- $read \in N \to 2^V$ *gives per node the set of read variables*
- $write \in N \to 2^V$ *gives per node the set of written variables*

For the configurable data perspective, the user has to option to change the relation between the entities. On the outgoing edges of choice constructs, there is the option to select an expression from one or more expressions. For variables, we can change which nodes read/write this variable.

Definition 8 (Configurable data perspective). *Let* $P_{cf} = (N, E, blockable, hideable, r, order)$ *be a configurable control-flow perspective, then a configurable data perspective is an 8-tuple* $(V, Ex, G, read, write, R_G, R_{read}, R_{write})$ *where:*

- V *is the set of variables*
- $Ex \subseteq \mathcal{V} \cup \{NoExpression\}$ *is the set of expressions, where* $\mathcal{V}$ *is the universe of logical expressions over* V*, and* $\{NoExpression\}$ *denotes that no expression is specified*
- $G \in E \to 2^{Ex}$ *annotates edges with expressions guarding the execution*
- $read \in N \to 2^V$ *gives per node the set of read variables*
- $write \in N \to 2^V$ *gives per node the set of written variables*
- $R_G \subseteq E \times Ex$ *gives the set of removable expressions from edges*
- $R_{read} \subseteq N \times V$ *gives the set of removable read variables from nodes*
- $R_{write} \subseteq N \times V$ *gives the set of removable write variables from nodes*

Configuration A configuration for the data perspective entails specifying which expression are removed/kept, which variables are read/not read, and which variables are written/ not written.

Definition 9 (Data configuration). *Let* $P_{cd} = (V, Ex, G, read, write, R_G, R_{read}, R_{write})$ *be a configurable data perspective, then a data configuration is a 6-tuple* $(r_G, \overline{r_G}, r_{read}, \overline{r_{read}}, r_{write}, \overline{r_{write}})$ *where:*

- $r_G \subseteq R_G$*, the set of removed expressions*
- $\overline{r_G} \subseteq R_G$*, the set of non-removed expressions*
- $r_{read} \subseteq R_{read}$*, the set of removed read variables*
- $\overline{r_{read}} \subseteq R_{read}$*, the set of non-removed read variables*
- $r_{write} \subseteq R_{write}$*, the set of removed written variables*
- $\overline{r_{write}} \subseteq R_{write}$*, the set of non-removed written variables*

Definition 10 (Valid data configuration). *Let* $P_{cd} = (V, Ex, G, read, write, R_G, R_{read}, R_{write})$ *be a configurable data perspective, and let* $C_d = (r_G, \overline{r_G}, r_{read}, \overline{r_{read}}, r_{write}, \overline{r_{write}})$ *be a data configuration, then* C_d *is valid if and only if:*

- $r_G \cap \overline{r_G} = \emptyset$*, a guard is not both selected and not selected*
- $\forall_{(e,ex),(e',ex') \in \overline{r_G}} (e = e' \Rightarrow ex = ex')$*, we only allow a single expression to be selected for an edge*
- $\forall_{e \in E} (\{(e, ex) \mid ex \in G(e)\} \setminus r_G \neq \emptyset)$*, we do not allow to remove all the possible expressions from an edge*
- $r_{read} \cap \overline{r_{read}} = \emptyset$*, a read variable cannot both be removed and not removed*
- $r_{write} \cap \overline{r_{write}} = \emptyset$*, a write variable cannot both be removed and not removed*

Definition 11 (Total data configuration). *A configuration* $C_d = (r_G, \overline{r_G}, r_{read}, \overline{r_{read}}, r_{write}, \overline{r_{write}})$ *is total for a configurable data perspective* $P_{cd} = (V, Ex, G, read, write, R_G, R_{read}, R_{write})$ *if and only if:*

- $(r_G \cup \overline{r_G}) = R_G$
- $(r_{read} \cup \overline{r_{read}}) = R_{read}$
- $(r_{write} \cup \overline{r_{write}}) = R_{write}$

4.3 Resource perspective

The resource perspective specifies which originators are present, and which manual task may be executed by which originator, e.g., in Fig. 2, we have an originator r_1 and she may execute D and E. We have chosen to call our resources originators based on the MXML standard.

Definition 12 (Originator perspective). *Let* $P_{cf} = (N, E, blockable, hideable, r, order)$ *be a configurable control-flow perspective, then an originator perspective is a 2-tuple* $(O, originators)$ *where:*

- O *is the set of originators*
- $originators : N_M \rightarrow (2^O \setminus \emptyset)$ *gives the non-empty set of originators for a manual task*

A configurable originator perspective denotes which originators are removable from allocatability to a manual task.

Definition 13 (Configurable resource perspective). *Let* $P_{cf} = (N, E, blockable, hideable, r, order)$ *be a configurable control-flow perspective, then a configurable originator perspective is a 3-tuple* $(O, originators, R_{originators})$ *where:*

- O *is the set of originators*
- $originators : N_M \rightarrow (2^O \setminus \emptyset)$ *gives the non-empty set of originators for a manual task*
- $R_{originators} \subseteq N_M \times O$ *gives the set of removable originators from a manual task*

Configuration A configuration for the originator perspective specifies which originator are or are not allocatable to a manual task.

Definition 14 (Originator Configuration). *Let* $P_{co} = (O, originators, R_{originators})$ *be a configurable originator perspective, then an originator configuration is a 2-tuple* $(r_{originators}, \overline{r_{originators}})$ *where:*

- $r_{originators} \subseteq R_{originators}$*, the set of removed originators from a manual task*
- $\overline{r_{originators}} \subseteq R_{originators}$*, the set of non-removed originators from a manual task*

A configuration is valid if the sets of (non) removed originators is disjoint, and every manual task has at least one originator which may execute that task.

Definition 15 (Valid originator configuration). *Let* $P_{co} = (O, originators, R_{originators})$ *be a configurable originator perspective, and let* $C_o = (r_{originators}, \overline{r_{originators}})$ *be an originator configuration, then* C_o *is total if and only if:*

- $r_{originators} \cap \overline{r_{originators}} = \emptyset$*, an originator cannot both be removed and not removed*
- $\forall_{a \in N_M} (\{(a, o) \mid o \in originators(a)\} \setminus r_{originators} \neq \emptyset)$*, we cannot remove all the originators from a manual task*

A total originator perspective configuration entails that for every removable originator the user has specified if she should be removed or not.

Definition 16 (Total originator configuration). *A configuration* $C_o = (r_{originators}, \overline{r_{originators}})$ *is total for a configurable originator perspective* $P_{co} = (O, originators, R_{originators})$ *if and only if:*

- $(r_{originators} \cup \overline{r_{originators}}) = R_{originators}$

4.4 Environment perspective

Currently, we only have properties in the environment perspective, these are treated later.

4.5 Experiment perspective

Currently, we only have properties in the experiment perspective, these are treated later.

4.6 Process Tree

A Process Tree is a combination of all the aforementioned perspectives and a generic labelling function. We use Σ as a shorthand for the set of characters. The Process Tree can formally be defined as:

Definition 17 (Process Tree). *Let $P_{cf} = (N, E, blockable, hideable, r, order)$ be a configurable control-flow perspective, let $P_{cd} = (V, Ex, G, read, write, R_G, R_{read}, R_{write})$ be a configurable data perspective for P_{cf}, let $P_{co} = (O, originators, R_{originators})$ be a configurable originator perspective for P_{cf}, let P_{ce} be a configurable environment perspective for P_{cf}, and let P_{cx} be a configurable experiment perspective for P_{cf}, then a Process Tree is a 6-tuple $PT = (label, P_{cf}, P_{cd}, P_{co}, P_{ce}, P_{cx})$ where:*

- *$label \in (PT \cup N \cup E \cup O \cup V \cup Ex) \rightarrow \Sigma^+$ is a labelling function*
- *P_{cf} is a configurable control-flow perspective*
- *P_{cd} is a configurable data perspective*
- *P_{co} is a configurable originator perspective*
- *P_{ce} is a configurable environment perspective*
- *P_{cx} is a configurable experiment perspective*

The meta model for the Process Tree is depicted in Fig. 5. The parts in red are related to the control-flow perspective, the originator perspective is in green, and the data perspective is in blue.

Having our process model formalism in place, we can use it in our framework. In the remainder, whenever the term process model is used, we intend a Process Tree.

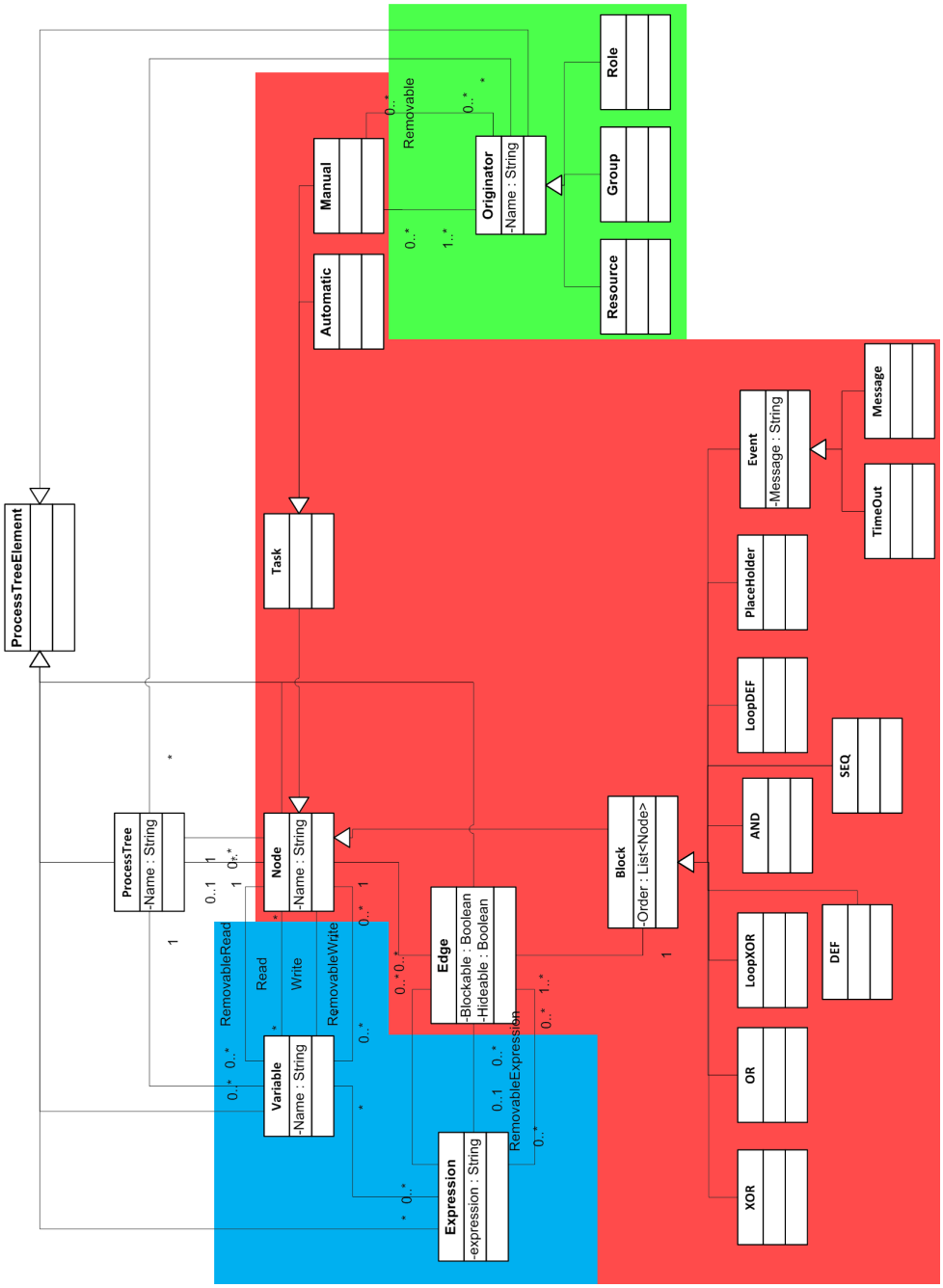

Fig. 5: The meta model of the Process Tree, red is control-flow perspective, green is originator perspective, and blue is data perspective.

5 Properties

We first introduce the notion of a (configurable) property as these are the basis of all the properties in this section. A property specifies a domain, e.g., the naturals, a comparator (comp) to denote which value is better, an identity value which can be used when a value for this property is missing, and a set of constructs on which this property has semantics.

The comparator is defined using $\leftarrow, \leftrightarrow, \rightarrow$ instead of $<, =, >$ since $=$ is ambiguous, e.g., when comparing distributions we might not have equality, but they are also not significantly different. The constructs are the constructs from the Process Tree, e.g., nodes, the Process Tree itself, variables, etc.

As mentioned, properties come in two different flavours, independent and dependent. In general, independent properties do not have an ordering, i.e., they are facts. We have combined independent and dependent properties into a single generic notion of a property. This is done to allow properties to be used both dependent and independent, but also on a conceptual level there is no difference, i.e., they describe characteristics of a construct.

Definition 18 (Property). *A property is a 4-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr})$ *where:*

- $\mathcal{D}om$ *is typing the domain of the property*
- $comp : \mathcal{D}om \times \mathcal{D}om \rightarrow \{\leftarrow, \leftrightarrow, \rightarrow\}$ *is a comparator, where the arrow points to the best value (possibly both values)*
- $identity$ *is the default value*
- $\mathcal{T}_{constr}$ *specifies for which constructs this property has semantics, e.g., the number of originators only makes sense for originators*

The configurable property specifies which values may be removed, i.e., which values this property cannot take. This works since we are reasoning from a closed world assumption for properties, i.e., beforehand the values a property may take are specified.

Definition 19 (Configurable property). *A configurable property is a 5-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *where:*

- $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr})$ *is a property*
- $removable \subseteq \mathcal{D}om$ *denoting the elements removable from the domain*

A configuration for a property entails specifying which removable values are removed and which are maintained.

Definition 20 (Configuration for a property). *Let* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *be a configurable property, then a configuration for a configurable property is a 2-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *where:*

- $removed \subseteq removable$*, the set of removed values*

– $\overline{removed} \subseteq removable$, *the set of non-removed values*

A valid configuration denotes that we not not remove all of the values. Note that some configurable properties have an added constraint to a valid configuration. When this is the case, there is a definition at that property.

Definition 21 (Valid configuration property). *Let* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *be a configurable property, and let* $C_p = (removed, \overline{removed})$ *be a configuration for a property, then* C_p *is valid if and only if:*

– $removed \cap \overline{removed} = \emptyset$ *a value cannot both be removed and not removed*
– $\mathcal{D}om \setminus removed \neq \emptyset$ *we do not remove all of the possible values*

If for each removable value we have specified if it is removed or maintained, we have a total configuration for a property. Note that, by applying a total configuration to a configurable property, we transform a configurable property to a non-configurable property. In the remainder of this document, if the $\mathcal{D}om$ of a configurable property is the same as the $\mathcal{D}om$ of the non-configurable variant of that property, we omit the definition of the property itself as this is the same except for the *removable* element. If this is not the case, we list both, e.g., for the initial value. The initial value of a variable can have a single value, but in order to make it configurable one needs to be able to choose from a set of variables.

Taking 2, we have added some properties to their respective perspective in Fig. 6 to show a more complete picture. The removable values are denoted with a cross.

5.1 Control-flow perspective

The control-flow perspective does not have at the moment any property associated to it.

5.2 Data perspective

For the data perspective, we currently have properties for variables. The properties of a variable consist of which data values that variable may take, what the next value for a variable is, and the initial value for a variable. In Fig. 7, the properties for a variable are encoded in a single figure. For instance, the variable v_1 has initial value "0", it may take the values "0", "1", and "2", and it changes value according to the probabilities on the edges.

Data Values property The data values property specifies which values a variable may take. The different values are all literals thus the values do not need to come from the same domain, e.g., a variable may take the values $\{0, a, X\}$. These values allow for simulating real writes to, and reads from variables. By having a predetermined set of values, also guards can reason on the different values a variable may take.

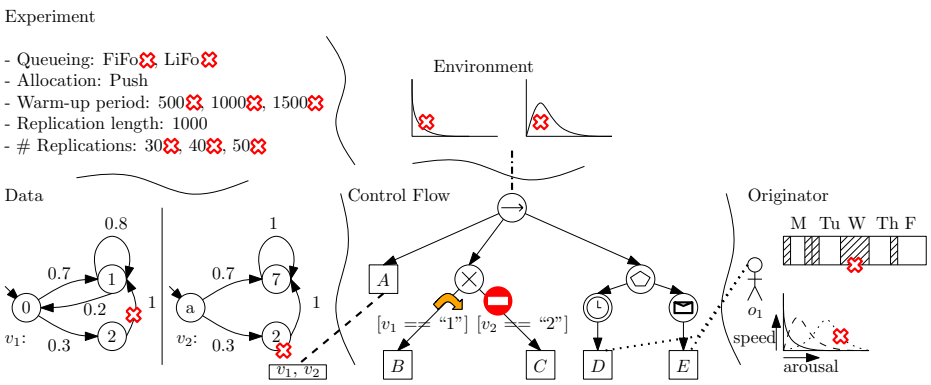

Fig. 6: The different perspectives in a Process Tree. The experiment perspective specifies the settings for the experiment. The environment perspective allows the user to specify the arrival process. The data perspective specifies the initial value for a variable, which values a variable may take, and the probability of changing one value to another value. The control-flow perspective specifies the execution sequence of activities. Finally, the originator perspective specifies when originators are available (M is for Monday, etc.) and what their working speed is (different distributions).

Definition 22 (Configurable Data Values property). *A configurable data values property is a 5-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *where:*

- $\mathcal{D}om \subseteq \Sigma^+$ *the set of values possible for a variable*
- $comp(d \in \mathcal{D}om, d' \in \mathcal{D}om) = \leftrightarrow$
- $identity = \emptyset$
- $\mathcal{T}_{constr} = \{V\}$*, the set of variables*
- $removable \subseteq \mathcal{D}om$ *the set of values a variable may not take*

In Fig. 7, we have the option to remove the value 2 for variable v_2. By selecting this option, we obtain the possible values (states) depicted in (a) for v_2. If we do not select this option, we obtain (b).

Data Value Matrix property The data value matrix specifies what the probability is for a variable to change its value. This is similar to a Markov-chain where each state in the Markov-chain is the possible values a variable can take. Moving from a current value to a new value is chosen probabilistically.

Definition 23 (Configurable Data Value Matrix property). *A configurable data value matrix property is a 5-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *where:*

- $\mathcal{D}om \subseteq \Sigma^* \times \Sigma^* \times \mathbb{R}$
- $comp(d \in \mathcal{D}om, d' \in \mathcal{D}om) = \leftrightarrow$
- $identity = \emptyset$

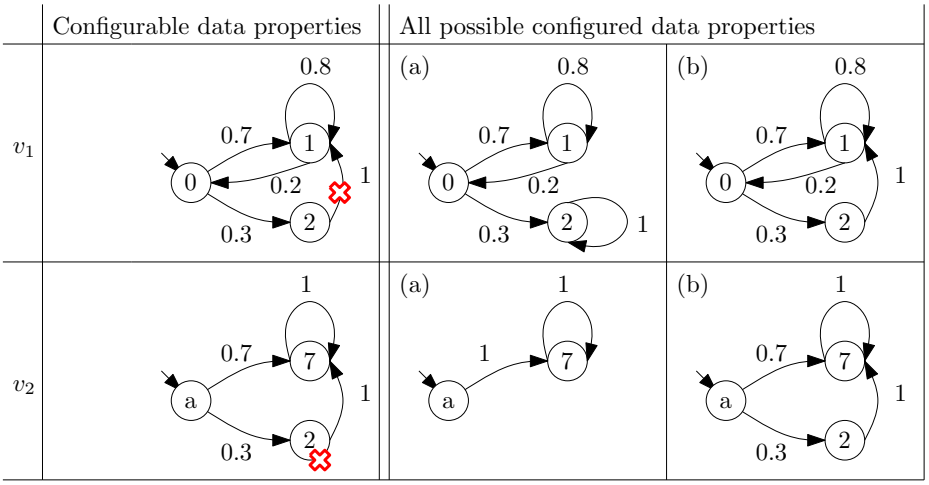

Fig. 7: The effect of setting the configuration options for a variable. The top row is for variable v_1, the bottom row for variable v_2, left of the double bar, the variable is configurable, and right of the double bar, the different configured variants for a variable are shown.

- $\mathcal{T}_{constr} = \{V\}$, *the set of variables*
- *removable* $\subseteq \mathcal{D}om$

In Fig. 7, we have the option to remove the edge from state 2 to state 1 for variable v_1. In (a) for v_1, we have selected to remove the edge, in (b) we have not selected to remove the edge.

Initial Value property The initial value specifies which of the possible values a variable may take is the initial value at the start of a case for a variable.

Definition 24 (Initial Value property). *An initial value property is a 4-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr})$ *where:*

- $\mathcal{D}om \in \Sigma^*$
- $comp(d \in \mathcal{D}om, d' \in \mathcal{D}om) = \leftrightarrow$
- $identity =$ `UNKNOWN`
- $\mathcal{T}_{constr} = \{V\}$, *the set of variables*

Definition 25 (Configurable Initial Value property). *A configurable initial value property is a 5-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *where:*

- $\mathcal{D}om \subseteq \Sigma^*$
- $comp(d \in \mathcal{D}om, d' \in \mathcal{D}om) = \leftrightarrow$
- $identity =$ `UNKNOWN`
- $\mathcal{T}_{constr} = \{V\}$, *the set of variables*
- *removable* $\subseteq \mathcal{D}om$

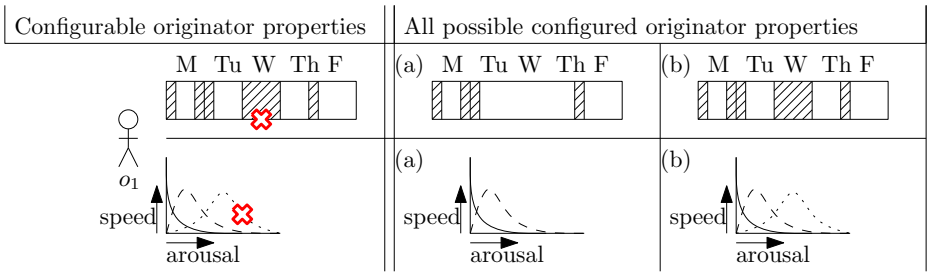

Fig. 8: The effect of modifying the work schedule and the arousal based work speed. The configurable variants are left of the double bar. On the right of the double bar, the configured variants are shown. Note that, the arousal based work speed is defined here for a single interval.

Since the value of the initial value property can only be a single value, we need to guarantee in our configuration that we keep at most 1 value.

Definition 26 (Valid configuration Initial Value property). *Let* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *be a configurable initial value property, and let* $C_p = (removed, \overline{removed})$ *be a configuration, then* C_p *is valid if and only if:*

- $removed \cap \overline{removed} = \emptyset$ *a value cannot both be removed and not removed*
- $\mathcal{D}om \setminus removed \neq \emptyset$ *we do not remove all of the possible values*
- $|\overline{removed}| \leq 1$, *we keep at most 1 value*

5.3 Originator perspective

The properties of the originator perspective specify (1) when an originator is available to work on the process (closely related and inspired by [6]), e.g., o_1 is available on Monday morning and evening, but not on Friday (Fig. 8), and (2) how many originators there are of a particular kind. Finally, different working speeds can be specified based on the busyness of an originator, this to model effects such as the phenomenon described by the Yerkes-Dodson law of arousal [35].

Number of originators This property denotes the number of originators of a particular role/group/resource are available to the process.

Definition 27 (Number of Originators property). *A number of originators property is a 4-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr})$ *where:*

- $\mathcal{D}om \in \mathbb{N}$
- $comp(d \in \mathcal{D}om, d' \in \mathcal{D}om) = \leftrightarrow$
- $identity = 0$
- $\mathcal{T}_{constr} = \{O\}$, *the set of originators*

Definition 28 (Configurable Number of Originators property). *A configurable number of originators property is a 5-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *where:*

- $\mathcal{D}om \subseteq \mathbb{N}$
- $comp(d \in \mathcal{D}om, d' \in \mathcal{D}om) = \leftrightarrow$
- $identity = 0$
- $\mathcal{T}_{constr} = \{O\}$*, the set of originators*
- $removable \subseteq \mathcal{D}om$

Since the value of the number of originators property can only be a single value, we need to guarantee in our configuration that we keep at most 1 value.

Definition 29 (Valid configuration Number of Originators property). *Let* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *be a configurable number of originators property, and let* $C_p = (removed, \overline{removed})$ *be a configuration, then* C_p *is valid if and only if:*

- $removed \cap \overline{removed} = \emptyset$ *a value cannot both be removed and not removed*
- $\mathcal{D}om \setminus removed \neq \emptyset$ *we do not remove all of the possible values*
- $|\overline{removed}| \leq 1$*, we keep at most 1 value*

Work schedule The work schedule consists of pairs of intervals, where the first element denotes the presence to the process and the second the absence from the process. For an increase of usability, we allow for different granularities of time, e.g., an originator may be present for 4 hours, and then absent for 1 day. The work schedule can be used to mimic the behaviour of part-time employees or employees not full-time available to the process.

Definition 30 (Configurable Work Schedule property). *A configurable work schedule property is a 5-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *where:*

- $\mathcal{D}om \subseteq \left((\mathbb{R} \times \mathbb{R} \times \gamma)^2\right)^*$*, the list of available, unavailable times, and* γ *specifying the time granularity*
- $comp(d \in \mathcal{D}om, d' \in \mathcal{D}om) = \leftrightarrow$
- $identity = \emptyset$
- $\mathcal{T}_{constr} = \{O\}$*, the set of originators*
- $removable \subseteq \mathcal{D}om$

When we remove an available, unavailable pair of times, we substitute this by unavailable. This way there is no difference between removing a single element and removing a pair of elements. Note that, we do not allow to work more than the configurable work schedule allows, i.e., the sum of the intervals an originator is available is maximal in the configurable work schedule property.

In Fig. 8, we have the option to remove the availability of originator o_1 on Wednesday. If we select the option to remove the availability on Wednesday, we obtain (a), else we obtain (b).

Yerkes-Dodson law In psychology, there is a relation between the arousal level (amount of work), and the performance (speed at which a person is operating) called the Yerkes-Dodson law [35]. A common phenomena when a deadline is approaching and people start to work faster and faster. To accommodate Yerkes-Dodson, we allow for the specification of the processing speed of a particular originator executing a task when there is a number of work items allocatable to this originator. For instance, an originator might take 10 minutes when there are between 0 and 5 work items waiting, and takes 5 minutes when there are between 5 and 10 work items waiting.

We have chosen to annotate the manual tasks with the Yerkes-Dodson property, the reason for this is that first the control-flow perspective is configured. This way there is no need to go through every originator and check on existence of the manual tasks. Note that, $\mathcal{D}istr$ denotes the universe of distributions.

Definition 31 (Configurable Yerkes-Dodson property). *A configurable Yerkes-Dodson property is a 5-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *where:*

- $\mathcal{D}om \subseteq O \times (\mathbb{N} \times \mathbb{N} \times \mathcal{D}istr)^+$*, the list of intervals (for allocatable work items) with the corresponding work speed (distribution) for an originator*
- $comp(d \in \mathcal{D}om, d' \in \mathcal{D}om) = \leftrightarrow$
- $identity = \emptyset$
- $\mathcal{T}_{constr} = \{N_A\}$*, the set of manual tasks*
- $removable \subseteq \mathcal{D}om$

In Fig. 8, we have the option to remove a distribution for this particular interval. The effects of choosing one of the options is depicted in (a) and (b).

5.4 Environment perspective

Currently, we only have the arrival process as property for the environment perspective. The arrival process specifies the distribution of arrivals of new cases to the process.

Arrival Process property The arrival process property specifies the arrival process of cases for this process (Fig. 6 at the centre top). We have chosen predetermined distributions for the ease of specifying constraint over the values specific for distributions, i.e., by explicitly enumerating the different distributions possible there is no risk of allowing a distribution not intended by the designer. Note that, $\mathcal{D}istr$ denotes the universe of distributions.

Definition 32 (Arrival Process property). *An arrival process property is a 4-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr})$ *where:*

- $\mathcal{D}om \in \mathcal{D}istr$
- $comp(d \in \mathcal{D}om, d' \in \mathcal{D}om) = \leftrightarrow$
- $identity =$ `Constant(0 milliseconds)`*, the constant distribution with an interarrival time of 0 milliseconds*

– $\mathcal{T}_{constr} = \{PT\}$, *the Process Tree itself*

Definition 33 (Configurable Arrival Process property). *A configurable arrival process property is a 5-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *where:*

– $\mathcal{D}om \subseteq \mathcal{D}istr$
– $comp(d \in \mathcal{D}om, d' \in \mathcal{D}om) = \leftrightarrow$
– $identity =$ `Constant(0 milliseconds)`, *the constant distribution with an interarrival time of 0 milliseconds*
– $\mathcal{T}_{constr} = \{PT\}$, *the Process Tree itself*
– $removable \subseteq \mathcal{D}om$

Since the value of the arrival process property can only be a single value, we need to guarantee in our configuration that we keep at most 1 value.

Definition 34 (Valid configuration Arrival Process property). *Let* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *be a configurable arrival process property, and let* $C_p = (removed, \overline{removed})$ *be a configuration, then* C_p *is valid if and only if:*

– $removed \cap \overline{removed} = \emptyset$ *a value cannot both be removed and not removed*
– $\mathcal{D}om \setminus removed \neq \emptyset$ *we do not remove all of the possible values*
– $|\overline{removed}| \leq 1$, *we keep at most 1 value*

5.5 Experiment perspective

The experiment perspective currently consists of the simulation property.

Simulation property The simulation property specifies the arguments for the simulation tool (Fig. 6 top left). $\mathcal{Q}$ denotes the chosen queue type, e.g., FiFo. The boolean denotes if push or pull allocation is used (true for push allocation, and false for pull allocation). The numbers represent warm-up period, replication length, and number of replications. Note that the values chosen for the identity function are chosen just to adhere to the requirements on a property and do not guarantee good experiments.

Definition 35 (Simulation property). *An simulation property is a 4-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr})$ *where:*

– $\mathcal{D}om \in \mathcal{Q} \times \mathbb{B} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$
– $comp(d \in \mathcal{D}om, d' \in \mathcal{D}om) = \leftrightarrow$
– $identity =$ `(FiFo, true, 0, 10, 100)`
– $\mathcal{T}_{constr} = \{PT\}$, *the Process Tree itself*

Definition 36 (Configurable Simulation property). *A configurable simulation property is a 5-tuple* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *where:*

– $\mathcal{D}om \subseteq \mathcal{Q} \times \mathbb{B} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$

- $comp(d \in \mathcal{D}om, d' \in \mathcal{D}om) = \leftrightarrow$
- $identity =$ `(FiFo, true, 0, 10, 100)`
- $\mathcal{T}_{constr} = \{PT\}$, *the Process Tree itself*
- $removable \subseteq \mathcal{D}om$

Since the value of the simulation property can only be a single value, we need to guarantee in our configuration that we keep at most 1 value.

Definition 37 (Valid configuration Simulation property). *Let* $(\mathcal{D}om, comp, identity, \mathcal{T}_{constr}, removable)$ *be a configurable simulation property, and let* $C_p = (removed, \overline{removed})$ *be a configuration, then* C_p *is valid if and only if:*

- $removed \cap \overline{removed} = \emptyset$ *a value cannot both be removed and not removed*
- $\mathcal{D}om \setminus removed \neq \emptyset$ *we do not remove all of the possible values*
- $|\overline{removed}| \leq 1$, *we keep at most 1 value*

6 Sample tool: CPN Tools

In this section, we elaborate on the transformation of a Process Tree to a CPN model. In Sect. 7, we mention the implementation necessarily to control CPN Tools, i.e., using Access/CPN, and parsing the output.

6.1 Transforming a Process Tree to a CPN model

Transforming a Process Tree to a CPN model is done along four perspectives: control-flow, data, originators, and environment, the experiment perspective consists mainly of controlling CPN Tools itself. Each of the four perspectives is transformed orthogonal to the other, i.e., modifications only impacting a single perspective also only modify the CPN model in that perspective so there is no need to worry about the other perspectives. To be able to still allow for communication between the perspectives, we have encoded a number of controllers, e.g., the control-flow perspective can only continue if a work-item has an originator allocated to it. The used functions in the CPN model are listed in Appendix B.2.

Control-flow perspective Within the control flow, we have that certain parts of the process model do not always contribute to a case. For instance, with an OR construct. To prevent the explicit modelling of all possible paths through the OR construct, we have introduced the notion of true/false tokens. If the true/false token is true, then that part of the process contributes to the case, otherwise not [36].

To be able to cope with multiple cases, we add to each token the notion of a case identifier. Furthermore, this means that transitions for synchronising a subprocess, e.g., the parallel execution of a number of tasks, are now guarded with the requirement that they can only synchronise if the tokens of their children all belong the same case, i.e., has the same identifier. With the aforementioned

directed acyclic graph of the Process Tree, we might have interference between multiple instances of the same case. By unfolding the graph to a tree and the fact that a Process Tree has single entry/single exit blocks, we do not have any problems with multiple instances interfering with each other.

In order to obtain timing information of the execution of a case, we have extended every token with a timestamp, duration, and performance information about the sojourn time, processing time, queue time, and wait time. The timestamp denotes when a token entered a subprocess rooted at the block node. The duration denotes the processing time for a task and is updated just before the tasks starts. We have separated the computation of the duration from the control-flow perspective as this is more flexible, e.g., in the future, one might want to change the way the duration is determined while the control flow does not change. Finally, the performance information is used to obtain information about subprocesses and can be used to compute the performance characteristics of a node based on its children. We measure the sojourn time, i.e., the total time a case spent in a (sub)process, the queue time, i.e., the time a work item waited on an originator, the wait time, i.e., the time a case spent waiting to be synchronised, and the processing time, i.e., the actual work originators have performed on the activities. The formal definition of a token is:

Definition 38 (Token). *Let $\mathbb{N}$ be the universe of natural numbers, let $\mathcal{T}$ be the universe of timestamps, let $\mathbb{B}$ be the universe of booleans, let $\mathbb{R}$ be the universe of reals, then $Token \in \mathbb{N} \times \mathcal{T} \times \mathbb{B} \times \mathbb{R} \times (\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R})$. Denoting the case identifier, time of entering the subprocess of the innermost block node, whether this token is true or false, the duration of the next task, and the accumulated performance statistics consisting of; sojourn time, processing time, queue time, and wait time.*

To query individual elements of the token, we use π_k as a shorthand notation where k denotes the index. For instance, let t be a token, then $\pi_1 t$ results in the case identifier of t. With $\pi_5 t$ for a token t, we obtain the accumulated performance statistics which can again be queried with π, e.g., $\pi_1 \pi_5 t$ yields the accumulated sojourn time.

Nodes All nodes offer the same interface places to their parent, see Fig. 9. The interface places offered to the parent are: *scheduled*, *started*, *withdrawn*, and *completed*. Scheduled denotes that a subprocess may start. Started denotes that a subprocess actually has started. Withdrawn is specifically for accommodating events, in which case all events are scheduled at the same time and the event that starts first withdraws the others. Finally, completed denotes that the subprocess has finished its execution. The interface places are encoded using fusion places in CPN Tools. A fusion place is a group of places combined into a fusion group. If a place in a particular fusion group gets (un)marked, then all places in that fusion group get (un)marked.

Connected to the interface places are some standard transitions. The *withdraw* transition fires when a token is received on interface place withdrawn, and

we have not started the execution of this subprocess. If the subprocess has already been started, then the withdrawn token is forwarded with high priority to the children. By using priorities on the transitions, we guarantee that nothing ordinary can happen between obtaining and forwarding the withdrawn token. When a subprocess is scheduled with a true token, then first the *init* transition is fired. The init transition is linked to another fusion place which forms a hook for a controller to notice that a subprocess has been scheduled, and to take action if required, e.g., delay the execution to model the manual handover of work. Similar, we have a *final* transition to consume the token after a controller has been notified on the completion of this subprocess. Finally, there is a *skip* transition in case a false token is received in the scheduled place.

Prior to going into the tasks and blocks, we first show the root net which is the controller connecting the environment perspective to the root node, and specifically the controller for the arrival of cases.

Root net The root net is the controller connecting the arrival process of the environment perspective to the control-flow perspective. At the top of the root net, Fig. 10, is the communication to the arrival process. At the bottom of the root net, the communication with the root node is encoded using the aforementioned interface to the parent (Fig. 9). Finally, at the right the different statistics of the entire process are collected. These statistics are the statistics which are collected in the token.

Tasks As tasks do not have children, the interface to the parent behaves slightly different (Fig. 11). The main difference is that we cannot receive a withdraw token after starting the task, i.e., we can only start the task after all unprocessed withdraw tokens have been processed. In case we receive a true token, the task signals the controller of this task being scheduled. As soon as the task is allowed to start, a token is produced signalling that the initialisation has been finished, and the task starts and signals its parent by producing a token on the started interface place. Afterwards, the task obtains the duration, which was set by the controller, from the token, and a token is made available after the duration has elapsed. Finally, the task signals that certain variables were read and written, and signals the controller of its completion resulting in the controller releasing the originator. We have explicitly modelled the reading of a variable as this might be used in the future (e.g., analysis of race-conditions on the data).

The communication between the task and the allocation of originators to work items is done at the Task Controller (Fig. 12). After a task has signalled the task controller it can be started, the task controller adds the work item to the list of currently unallocated work items. As soon as a originator has been allocated to a work item, the task controller notifies the task and determines the processing time for the task based on the Yerkes-Dodson law, i.e., the controller computes the pressure for the originator and looks up the speed of that originator performing that task. Finally, after the task signals its completion, the originator used by the task is released.

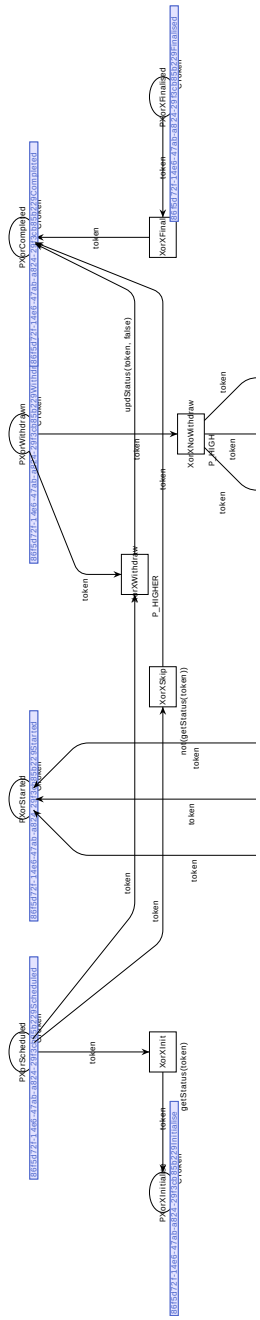

Fig. 9: The interface a node (XOR in this case) has towards its parent.

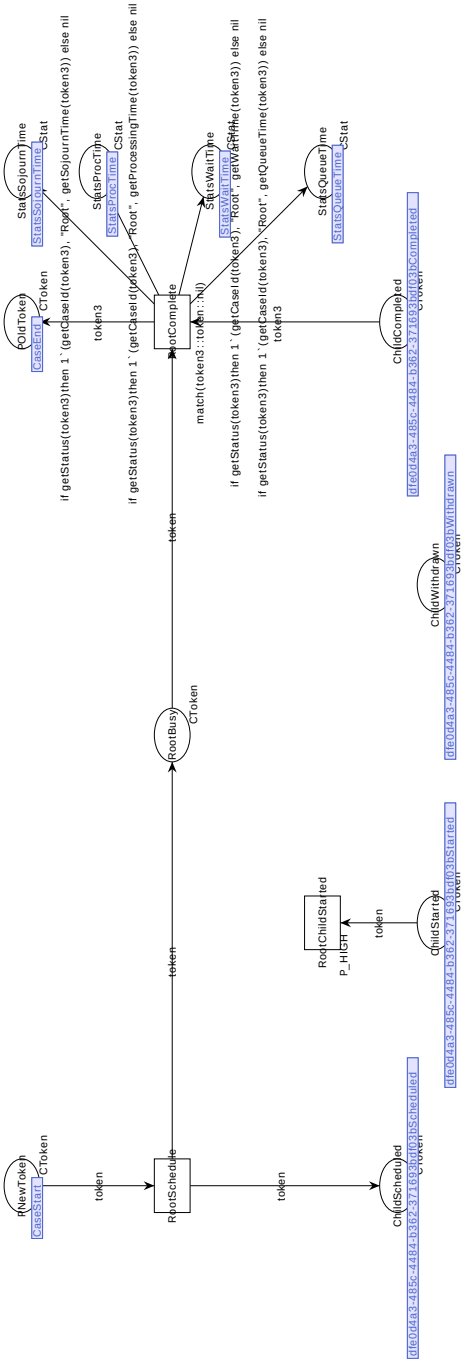

Fig. 10: The root net connecting the arrival of cases with the control flow.

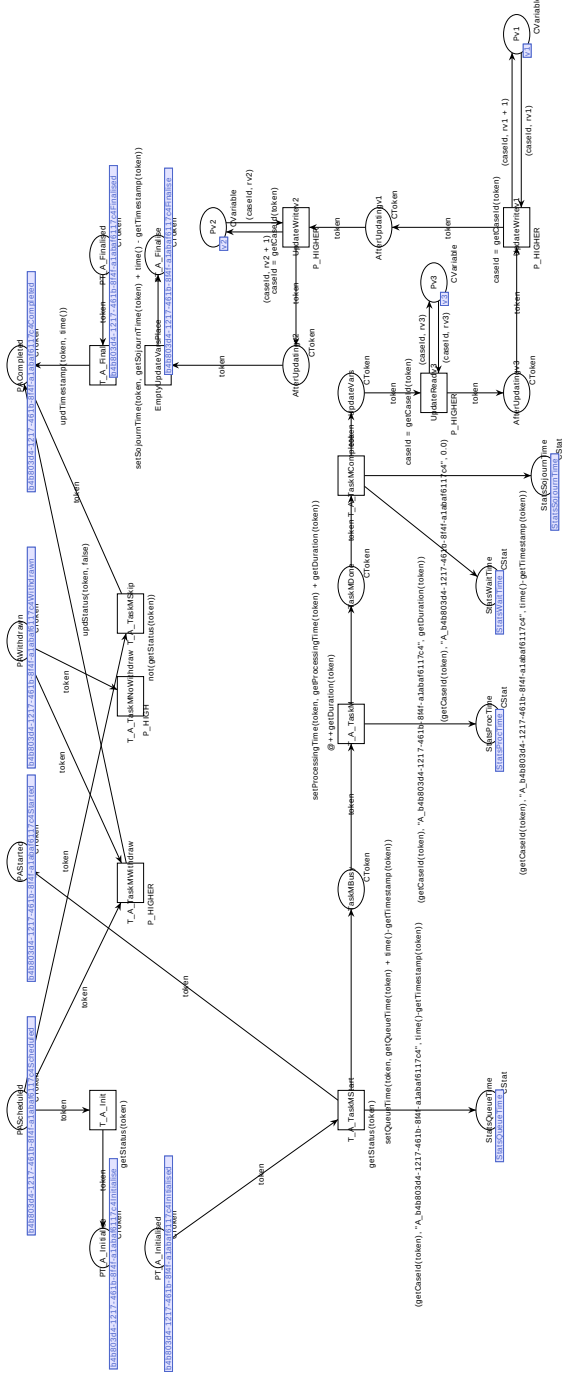

Fig. 11: Manual task encoding

Fig. 12: Task controller encoding

The statistics for a task are computed as follows; the queue time is the elapsed time between a token entering the task's subprocess and the moment we obtain the token back from the controller, i.e., an originator has been allocated to the work item. The processing time is the duration of a task. The wait time is 0 by definition. Finally, the sojourn time is the sum of the queue time and the processing time, i.e., the total time a token spent in the task subprocess. More formally:

Definition 39 (Statistics task). *Let d be the duration as determined by the controller, let q be the time spent inside the controller for allocating an originator (Core Controller), then the statistics are computed as follows:*

- *Sojourn time:* $d + q$
- *Processing time:* d
- *Queue time:* q
- *Wait time:* 0

Blocks Blocks have the same set of places exposed to their parent as a task has, e.g., also a block is scheduled, and a block signals her parent that she is completed. Dependent on the type of block, different children receive a true token based on different information, e.g., in case of an AND block all the children receive a true token, in case of a XOR block only the first child for which the expression evaluates to true receives a true token.

Blocks come in 8 different flavours and each of them is described in the remainder of this paragraph. In the figures of the blocks, we have placed the functions for updating the token and computing the sojourn, processing, queue, and wait time in that order at a free space in the figure for readability.

SEQ The sequence operator schedules its children one-by-one. As soon as a child has finished, the next child may be scheduled. In Fig. 13 (note the positioning of the functions in the bottom middle), the encoding of the sequence operator is shown. The first step for a scheduled sequence is the notification of the controller. Afterwards, the first child is scheduled. Note that, extra hooks have been added to notify the controller that a particular child is scheduled, this way delays in handovers can be modelled. If the first child has started, then the entire sequence has started and the parent is signalled. Similar, if the sequence is withdrawn, then this token is forwarded to the first child of the sequence. After the first child completed, the second child is scheduled after notifying the controller. Finally, after the last child has completed, the statistics are computed for this sequence and stored, and the controller is notified of the completion of the sequence.

The statistics are computed as follows; the sojourn/ processing/ queue/ wait time of the subprocesses rooted at the SEQ block, is the sojourn/ processing/ queue/ wait time of the token returned by the last child of the SEQ block subtracted with the sojourn/ processing/ queue/ wait time of the token arriving to the SEQ block, i.e., the sojourn/ processing/ queue/ wait time of the case till this block. More formally:

Definition 40 (Statistics seq block). *Let $n_1, \ldots, n_k$ be the children of the* SEQ *block, and let $t_1, \ldots, t_k$ be the token returned by the children, i.e., n_1 returns token t_1, and finally, let t be the token obtained from the parent of the* SEQ *block, then the statistics are computed as follows:*

- *Sojourn time:* $\pi_1\pi_5 t_k - \pi_1\pi_5 t$
- *Processing time:* $\pi_2\pi_5 t_k - \pi_2\pi_5 t$
- *Queue time:* $\pi_3\pi_5 t_k - \pi_3\pi_5 t$
- *Wait time:* $\pi_4\pi_5 t_k - \pi_4\pi_5 t$

AND The AND operator schedules its children all at once. After signalling the controller of being scheduled, the AND block schedules all of its children (Fig. 31). Since we do not know which of the children will be the first to start, the start notifications of all children are forwarded to the parent. Similar, we forward a withdraw token to all children to notify them. Finally, the AND block can only complete when all of the children have completed. The performance statistics are written, and the controller is notified.

The statistics for the AND block are similar to the SEQ block, i.e., we take the summation of the statistics of the individual children. However, as these children do not have a causal relationship, we have to do the summation manually. Furthermore, the wait time of the AND block is different from the SEQ block. Note that with a decomposition of the AND operator into multiple AND operators, we count the wait time multiple times, i.e., the wait times in the AND operators happen simultaneously but are counted as sequential.

Definition 41 (Statistics and block). *Let $n_1, \ldots, n_k$ be the children of the* AND *block, and let $t_1, \ldots, t_k$ be the token returned by the children, i.e., n_1 returns token t_1, and finally, let t be the token obtained from the parent of the* AND*, then the statistics are computed as follows:*

- *Sojourn time:* $\max_{1\leq i\leq k}(\pi_1\pi_5 t_i) - \pi_1\pi_5 t$
- *Processing time:* $\sum_{1\leq i\leq k}(\pi_2\pi_5 t_i - \pi_2\pi_5 t)$
- *Queue time:* $\sum_{1\leq i\leq k}(\pi_3\pi_5 t_i - \pi_3\pi_5 t)$
- *Wait time:* $\max_{1\leq i\leq k}(\pi_1\pi_5 t_i) - \min_{1\leq i\leq k}(\pi_1\pi_5 t_i) + \sum_{1\leq i\leq k}(\pi_4\pi_5 t_i - \pi_4\pi_5 t)$

XOR The XOR block is very similar to the AND block. The main difference is in the scheduling of the children, lower left in Fig. 32. Similar to the AND block, we first signal the controller of the scheduling of the children, this allows the controller to, for instance, inspect the guards and take appropriate actions. Afterwards, the guards are evaluated one-by one, i.e., first the guard for the first child is evaluated, if it is true then a true token is passed to the first child and false tokens are passed to the remaining children. If the first guard is false, the second guard is evaluated and if true, then a true token is passed to the second child and false tokens are passed to the other children. Finally, if all the guards are false, then a true token is sent to the last child. This is the same as in the YAWL [32] semantics, where the last child is a default option, i.e., its guard always evaluates to true.

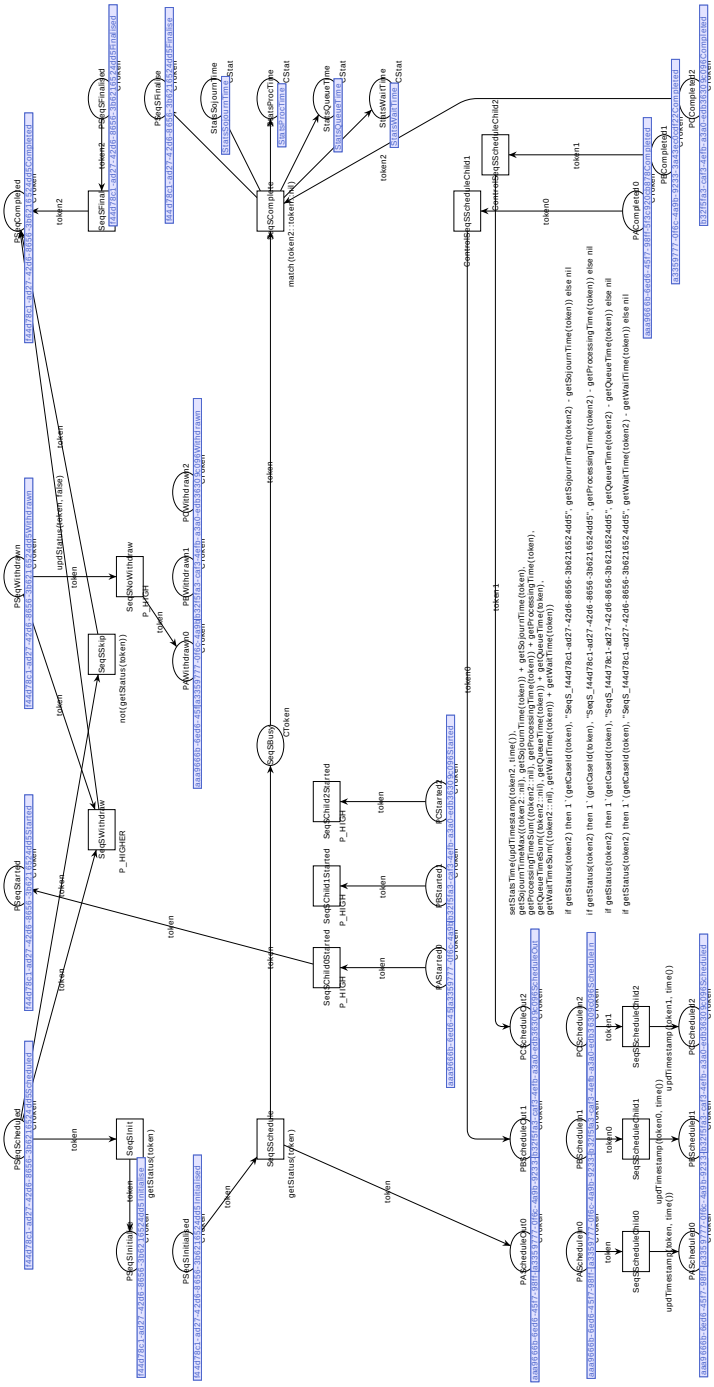

Fig. 13: SEQ operator encoding

The statistics for the XOR block are similar to the SEQ block, i.e., we take the summation of the statistics of the individual (executed) children which is a single one by definition. Since we have true and false token, we can use these to determine if a subprocess has been executed or not. Note that, we use a shorthand notation in the definition, e.g., let b be a boolean, then $5 \cdot b = 5$ in case b is true, and $5 \cdot b = 0$ otherwise.

Definition 42 (Statistics xor block). *Let $n_1, \ldots, n_k$ be the children of the* XOR *block, and let $t_1, \ldots, t_k$ be the token returned by the children, i.e., n_1 returns token t_1, and finally, let t be the token obtained from the parent of the* XOR*, then the statistics are computed as follows:*

- *Sojourn time:* $\max_{1 \leq i \leq k} (\pi_3 t_i \cdot \pi_1 \pi_5 t_i) - \pi_1 \pi_5 t$
- *Processing time:* $\sum_{1 \leq i \leq k} (\pi_3 t_i \cdot \pi_2 \pi_5 t_i - \pi_3 t_i \cdot \pi_2 \pi_5 t)$
- *Queue time:* $\sum_{1 \leq i \leq k} (\pi_3 t_i \cdot \pi_3 \pi_5 t_i - \pi_3 t_i \cdot \pi_3 \pi_5 t)$
- *Wait time:* $\sum_{1 \leq i \leq k} (\pi_3 t_i \cdot \pi_4 \pi_5 t_i - \pi_3 t_i \cdot \pi_4 \pi_5 t)$

LOOPXOR The LOOPXOR block is most similar to the SEQ block, since the LOOPXOR block can be seen as a sequence of *do*, *redo*, ..., *exit*. First, the *do* child is scheduled, and, when it has completed, both the *redo* and *exit* child are scheduled. Based on the evaluation of the guard of the *redo* child, see Fig. 14, either the *redo* or *exit* child gets the true token. Only the guard for the *redo* child is used since YAWL semantics are used, i.e., if the guard for the *redo* child is false, then by default the guard for the *exit* is true, hence the negation of the *redo* guard is used. If the *redo* child gets a true token, then the *do* child is executed again after synchronising with the false token from the *exit*. The LOOPXOR block terminates if the *exit* child has a true token and thus the *redo* child has a false token. Note that, we do not strictly need to use a false token for the non-selected child, this has been done for uniformity with the XOR block and LOOPDEF block.

The statistics for the LOOPXOR block are similar to the SEQ block, i.e., we take the statistics from the last child, i.e. the *exit* child.

Definition 43 (Statistics loopxor block). *Let $n_1, \ldots, n_3$ be the children of the* LOOPXOR *block, and let $t_1, \ldots, t_3$ be the token returned by the children, i.e., n_1 returns token t_1, and finally, let t be the token obtained from the parent of the* LOOPXOR*, then the statistics are computed as follows:*

- *Sojourn time:* $\pi_1 \pi_5 t_3 - \pi_1 \pi_5 t$
- *Processing time:* $\pi_2 \pi_5 t_3 - \pi_2 \pi_5 t$
- *Queue time:* $\pi_3 \pi_5 t_3 - \pi_3 \pi_5 t$
- *Wait time:* $\pi_4 \pi_5 t_3 - \pi_4 \pi_5 t$

OR The OR block has the collection of statistics from the AND block but taking into account which children have been executed, and the selection of the children is similar to the XOR block. The main difference with the XOR block is that now multiple children may be selected. If none of the guard for the children evaluates

Fig. 14: LOOPXOR operator encoding

Fig. 15: OR operator encoding

to true, then the last child is selected as the default option. The encoding of the OR block into CPN Tools is depicted in Fig. 15.

The statistics for the OR block are a hybrid between the AND block and the XOR block, i.e., we have similar formulas for the statistics, but we have to take the true token into account as mentioned with the XOR block. Since we only want to take the executed branches into account, we first filter the set of tokens to only contain the tokens of the executed branches (T). This is for the sake of readability. Note that, similar to the AND block, we might count the wait time multiple times if the OR block has been decomposed into multiple OR blocks.

Definition 44 (Statistics or block). *Let $n_1, \ldots, n_k$ be the children of the* OR *block, and let $t_1, \ldots, t_k$ be the token returned by the children, i.e., n_1 returns token t_1, and finally, let t be the token obtained from the parent of the* OR*, then the statistics are computed as follows:*

- $T : \{t_i \mid 1 \leq i \leq k \wedge \pi_3 t_i\}$
- *Sojourn time:* $\max_{t_i \in T} (\pi_1 \pi_5 t_i) - \pi_1 \pi_5 t$
- *Processing time:* $\sum_{t_i \in T} (\pi_2 \pi_5 t_i - \pi_2 \pi_5 t)$
- *Queue time:* $\sum_{t_i \in T} (\pi_3 \pi_5 t_i - \pi_3 \pi_5 t)$
- *Wait time:* $\max_{t_i \in T} (\pi_1 \pi_5 t_i) - \min_{t_i \in T} (\pi_1 \pi_5 t_i) + \sum_{t_i \in T} (\pi_4 \pi_5 t_i - \pi_4 \pi_5 t)$

EVENT Prior to showing the encoding of the DEF block and LOOPDEF block, we show the encoding of the EVENT block. The EVENT block starts as the other blocks by signalling the controller. Afterwards, the EVENT block is scheduled, and a token is produced in the *Pending* place, see Fig. 16. If another EVENT block is started before this EVENT block, then the *Pending* place is emptied, and the token from this EVENT block is removed. If no other EVENT block is started before this EVENT block, then its child is scheduled and the parent is signalled such that it can cancel the other pending events.

The statistics for an EVENT block are the same as the statistics of its child. The only difference is the sojourn time which also needs to contain the time until the the event occurred. Since the time until the the event occurred is added to the token before its child is started, we can simply use the difference in accumulated sojourn time between the token obtained for the parent and the token returned from its child.

Definition 45 (Statistics event block). *Let n_1 be the child of the* EVENT *block, and let t_1 be the token returned by the child, i.e., n_1 returns token t_1, and finally, let t be the token obtained from the parent of the* EVENT*, then the statistics are computed as follows:*

- *Sojourn time:* $\pi_1 \pi_5 t_1 - \pi_1 \pi_5 t$
- *Processing time:* $\pi_2 \pi_5 t_1 - \pi_2 \pi_5 t$
- *Queue time:* $\pi_3 \pi_5 t_1 - \pi_3 \pi_5 t$
- *Wait time:* $\pi_4 \pi_5 t_1 - \pi_4 \pi_5 t$

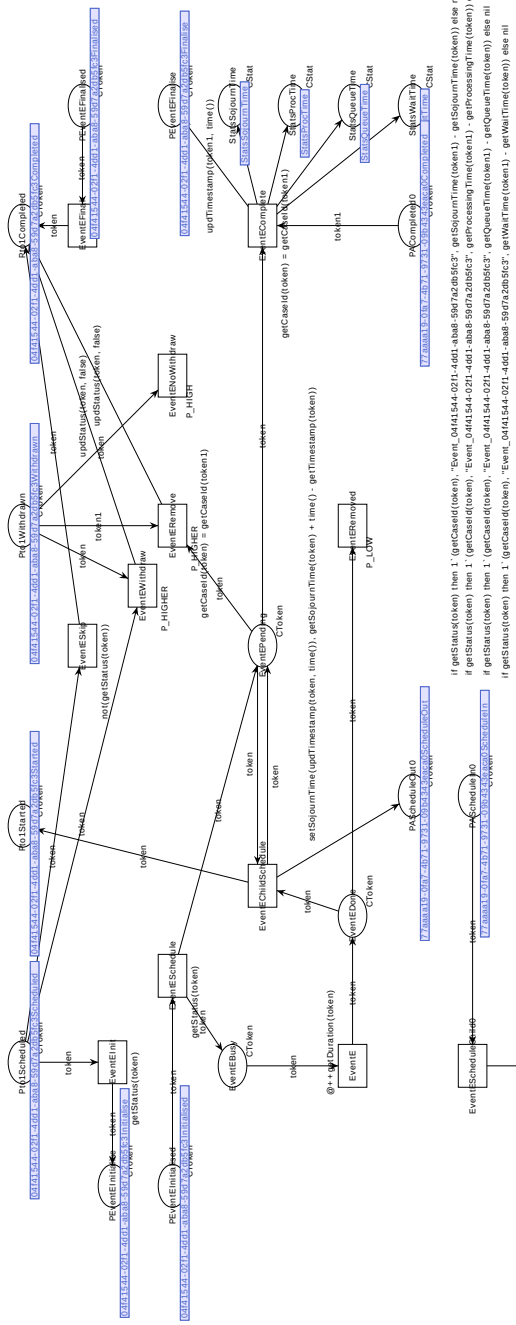

Fig. 16: EVENT operator encoding

DEF The DEF block is with respect to scheduling the children the same as an AND block, i.e., all the children are scheduled. However, as soon as one of the EVENT block children has started, then the other EVENT block children are withdrawn, centre of Fig. 36.

The statistics for the DEF block is the same as the statistics for the XOR block. We have repeated the definition for completeness.

Definition 46 (Statistics def block). *Let $n_1, \ldots, n_k$ be the children of the* DEF *block, and let $t_1, \ldots, t_k$ be the token returned by the children, i.e., n_1 returns token t_1, and finally, let t be the token obtained from the parent of the* DEF*, then the statistics are computed as follows:*

- *Sojourn time:* $\max_{1 \leq i \leq k} (\pi_3 t_i \cdot \pi_1 \pi_5 t_i) - \pi_1 \pi_5 t$
- *Processing time:* $\sum_{1 \leq i \leq k} (\pi_3 t_i \cdot \pi_2 \pi_5 t_i - \pi_3 t_i \cdot \pi_2 \pi_5 t)$
- *Queue time:* $\sum_{1 \leq i \leq k} (\pi_3 t_i \cdot \pi_3 \pi_5 t_i - \pi_3 t_i \cdot \pi_3 \pi_5 t)$
- *Wait time:* $\sum_{1 \leq i \leq k} (\pi_3 t_i \cdot \pi_4 \pi_5 t_i - \pi_3 t_i \cdot \pi_4 \pi_5 t)$

LOOPDEF The LOOPDEF operator is a combination of the LOOPXOR and the DEF blocks. Similar to the LOOPXOR block, as soon as the *do* child has finished, both the *redo* and *exit* children may start. However, instead of a true or false token, we activate both children and the EVENT block first to start its child cancels the other EVENT block child, e.g., if the *redo* event is first to start its child, then the *exit* event is cancelled, bottom centre of Fig. 37.

The statistics for the LOOPDEF block are the same as for the LOOPXOR block, i.e., there is no difference whether guards or events are used to guide the execution of the children.

Definition 47 (Statistics loopdef block). *Let $n_1, \ldots, n_3$ be the children of the* LOOPDEF *block, and let $t_1, \ldots, t_3$ be the token returned by the children, i.e., n_1 returns token t_1, and finally, let t be the token obtained from the parent of the* LOOPDEF*, then the statistics are computed as follows:*

- *Sojourn time:* $\pi_1 \pi_5 t_3 - \pi_1 \pi_5 t$
- *Processing time:* $\pi_2 \pi_5 t_3 - \pi_2 \pi_5 t$
- *Queue time:* $\pi_3 \pi_5 t_3 - \pi_3 \pi_5 t$
- *Wait time:* $\pi_4 \pi_5 t_3 - \pi_4 \pi_5 t$

Data perspective The data perspective comprises of variables and expressions.

Variables Variables come in 2 different views; control-flow view, and guard view. In Fig. 17, the control-flow view is at the top (fusion place *Pv1*) and guard view is at the bottom (fusion place *Pv1Value*). We have made the distinction because the different views serve two different purposes. For the control-flow view, it is irrelevant what the value of a variable is but it is important to know whether an update has been performed. Therefore, the control-flow view is a tuple of a case identifier, and a revision number. For the guard view, it is important to know the exact value of a variable as these are used to evaluate guards, the revision

of a variable is not important for the guard. For instance, if we have a variable *Assessment* denoting the assessment of a case, then in the control flow, when reading and writing the variable the value is not important. For the guard it is important what the value is but it is irrelevant how many times an update has been performed on this variable.

When a new case is started, a variable for the control-flow view is created with a new case identifier and revision number 0. In the variable controller, an *initialise* transition initialises a variable for the guard view with the variable's default value (0 in case of the example). Apart from initialising the variable for the guard view, we also store the current revision number of the variable. This revision number allows us to determine if a variable has been updated, i.e., the revision number changed. When the revision number changed, we use the *Value Matrix* to determine the next value for this variable. Furthermore, we store the new revision number of the variable. After the case has terminated, the case identifier is removed from the list of given identifiers (fusion place top right). As soon as a case identifier is removed, the stored revision number is removed as well as the variable for the guard view.

Expressions An expression is a single transition guarded by the expression itself. In Fig. 18, the expression `[(((v1 == "0") || (v2 == "1")) && (v3 == "4"))]` is encoded. Slightly different names for the variables are used as we are now interested in the value, i.e., the guard view. For the expression, all the used variables are inspected as well as the evaluation of the guard (true or false). If the guard of the transition is different from the stored evaluation, then we reevaluate the expression and write the new value (true or false). If the guard of the transition is not different, this means the evaluation of the expression did not change, hence there is no need to change the value.

Originator perspective The originator perspective mainly comprises of the work schedule of the originators. The number of originators available to the process is a modification of the initial marking.

We introduce a CPN property to be more flexible with storing more attributes for originators, e.g., in the future one might want to store more characteristics for a particular type of originator for instance whether a originator is a generalist or specialist.

Definition 48 (CPN Property). *Let Σ be an alphabet, then a CPN property is defined as: $\Sigma^+ \nrightarrow \Sigma^+$.*

This CPN property is used to determine the like factor between an originator and a work item (more on this later). Furthermore, the CPN property is used to encode role information of an originator. In Fig. 19, we have the fragment `("OrgName", "Role1")` denoting that this particular originator has *Role1* as *OrgName*.

The CPN schedule is a list of reals denoting the length of the intervals of being present and absent. The integer denotes at which index the schedule currently

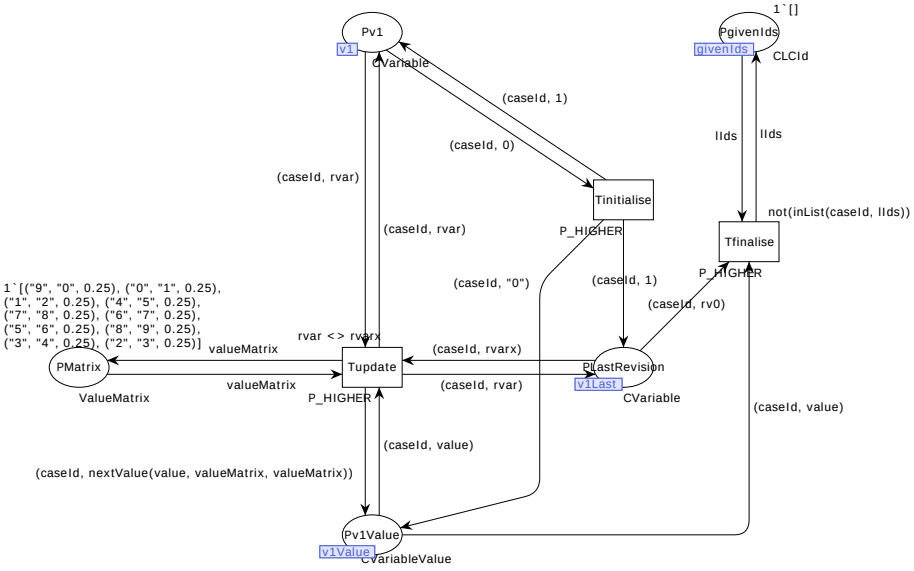

Fig. 17: Variable encoding

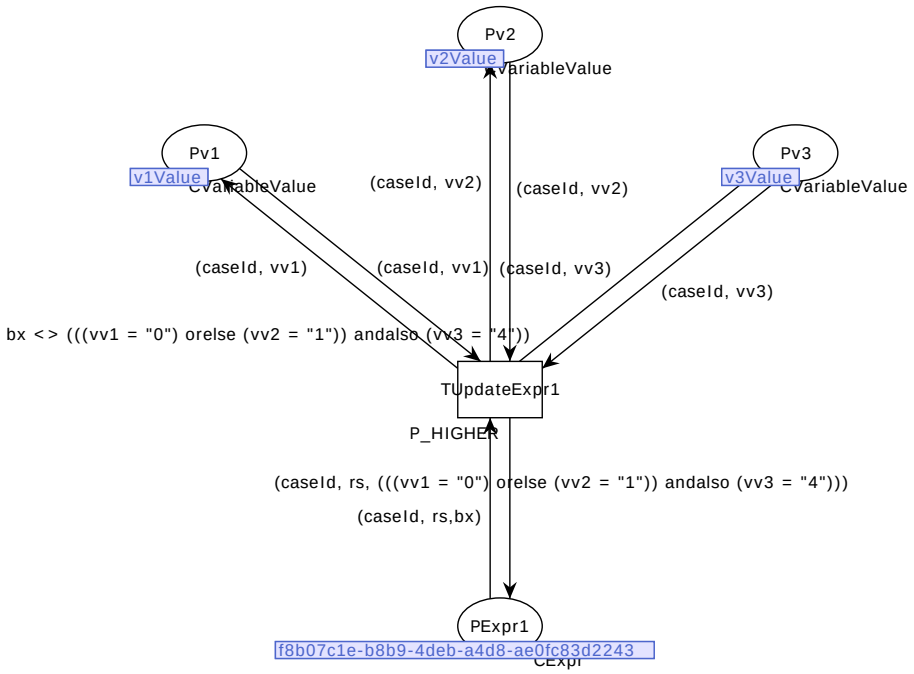

Fig. 18: Expression encoding

is, and the timestamp at the end denotes at what time the last change was. For instance, we have `(0, [0.20833333333333334, 0.08333333333333333, 0.2916666666666667, 0.25], 0.0)` in Fig. 19 on the first line of the marking. The "0" denotes that we are at index 0, i.e., the value "0.20833333333333334". The values between "[" and "]" are the encoding of the work schedule. Finally, the "0.0" at the end denotes the timestamp of the last change which in this case it was at time 0.0.

Definition 49 (CPN Schedule). *Let $\mathbb{N}$ be the universe of integers, let $\mathbb{R}$ be the universe of reals, and let $\mathcal{T}$ be the universe of timestamps, then a CPN schedule is defined as: $\mathbb{N} \times \mathbb{R}^+ \times \mathcal{T}$.*

A CPN originator has a name, a schedule, and a list of properties. These properties can be used to encode different characteristics of a particular originator.

Definition 50 (CPN Originator). *Let $\mathcal{S}$ be the universe of schedules, let $\mathcal{P}$ be the universe of CPN properties, then a CPN originator is defined as: $\Sigma^+ \times \mathcal{S} \times \mathcal{P}^*$.*

In Fig. 19, we have a list of originators and two control places as guards for the transitions. We need these control places since CPN Tools does not allow time related guards, i.e., guards which become enabled after some time has progressed but the marking did not change. The controller works as follows: after the time of the current index in the schedule has elapsed, the originator is made (un)available to the process and the index and timestamp are updated according. This keeps on continuing during the execution of the process.

6.2 Environment Perspective

The environment perspective currently consists of the arrival controller. The arrival controller is in charge of creating new cases and disposing of finished cases.

Arrival controller The arrival controller consists of a token generator, at the left hand-side of Fig. 20. This token generator signals to the root net that a new case has been started and adds the case identifier to the list of identifiers. Simultaneously, the expressions and variables are created with a standard initialisation, i.e., revision number 0. Due to historic reasons also the expressions have revision numbers but these are currently not used.

The right-hand side of the controller is charged with closing a case. As soon as the root net signals the case has been ended (a token is produced on *PCaseEnd*), the various expression places, and variable places associated to that case are removed. Finally, the case identifier is removed from the list of identifiers.

Controllers The controller not solely belonging to any of the aforementioned perspectives is the *core controller*. The core controller is in charge of the allocation of originators to work items.

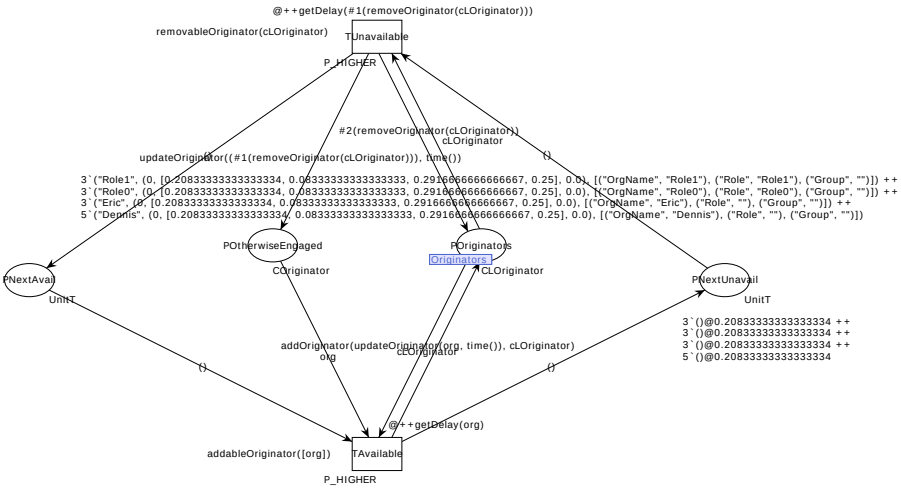

Fig. 19: The work schedule for the different originators.

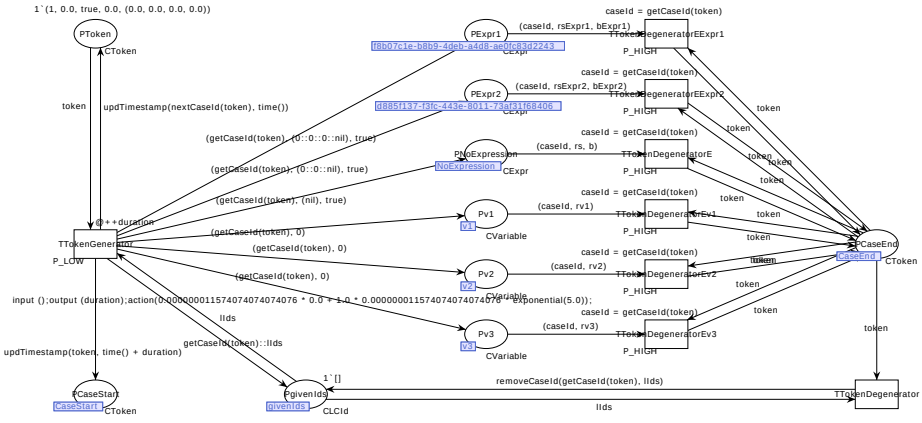

Fig. 20: The arrival controller responsible for the arrival process of cases.

Core Controller As mentioned, the core controller allocates work items to originators. It operates on the list of work items, which are cases waiting to be processed by an activity, and on the list of currently available originators to the process, see Fig. 21. The allocation can be either push or pull allocation by taking either the work item or originator point of view. For instance, if we take the originator point of view (pull), then the originator selects the work item most desired by her. When taking the work item point of view (push), we select the originator best fitting the work item.

Allocation of a work item to an originator, is a simple matching tuple, i.e., we store the originator which is going to execute a certain work item.

Definition 51 (Matching). *Let $\mathcal{W}$ be the universe of work items, let $\mathcal{R}$ be the universe of originators, then a matching is defined as: $m \in \mathcal{W} \times \mathcal{R}$.*

The actual allocation takes the list of work items currently allocated, the list of originators available, and the list of work items and creates a list of matchings.

Definition 52 (Originator Allocation). *Let $\mathcal{W}$ be the universe of work items, let $\mathcal{R}$ be the universe of originators, let $\mathcal{M}$ be the universe of matchings, then the matching function is defined as: $\mathcal{W}^* \times \mathcal{R}^* \times \mathcal{M}^* \to \mathcal{M}^*$*

In order to determine the allocation possibility between an originator and a work item, we have a *like* factor.

Definition 53 (Like factor). *Let $\mathcal{W}$ be the universe of work items, let $\mathcal{R}$ be the universe of originators, then the like factor $L_f : \mathcal{W} \times \mathcal{R} \to \mathbb{R}$.*

The different allocation strategies are now defined as:

Definition 54 (matchingPush). *Let W be a list of work items, let R be a list of originators, let M be a list of matchings, then matchingPush is defined as:*

$$
\begin{aligned}
matchingPush(W, R, M) &= matchingPush(W, R, M, R) \\
matchingPush([], [], M, R') &= M \\
matchingPush([], r :: R, M, R') &= M \\
matchingPush(w :: W, [], M, R') &= matchingPush(W, R', M, R') \\
matchingPush(w :: W, r :: R, M, R') &= (w, r) :: M && \text{if } L_f(w, r) \text{ is maximal} \\
matchingPush(w :: W, r :: R, M, R') &= matchingPush(w :: W, R, M, R') && \text{if } L_f(w, r) \text{ is not maximal}
\end{aligned}
$$

Definition 55 (matchingPull). *Let W be a list of work items, let R be a list of originators, let M be a list of matchings, then matchingPull is defined as:*

$$
\begin{aligned}
matchingPull(W, R, M) &= matchingPull(W, R, M, W) \\
matchingPull([], [], M, W') &= M \\
matchingPull([], r :: R, M, W') &= matchingPull(W', R, M, W') \\
matchingPull(w :: W, [], M, W') &= M \\
matchingPull(w :: W, r :: R, M, W') &= (w, r) :: M && \text{if } L_f(w, r) \text{ is maximal} \\
matchingPull(w :: W, r :: R, M, W') &= matchingPull(W, r :: R, M, W') && \text{if } L_f(w, r) \text{ is not maximal}
\end{aligned}
$$

Dependent on whether push or pull allocation is desired, we first iterate over the originators to find the best for a work item, or we first iterate over the work items to find the best for the first originator. If a match is possible, all lists are updated accordingly. That is, the work item and originator are removed from the lists of unallocated work items/originators, and the pair work item, originator are added to a list of allocated work items/originators.

The maximal like factor is determined beforehand. This is computed from either the work item or originator point of view based on push or pull allocation. With push allocation, we take a work item and check for each originator whether it is allocatable to this work item. In order to know wether an allocatable originator was encountered, we store the max like factor found so far (variable d). After checking every originator for a particular work item two possible outcomes are possible; (1) the maximal like factor was 0, which means none of the originator was allocatable, then we move to the next work item and do the above steps again. (2) the maximal like factor is larger than 0, which means there was an originator allocatable, and this value is returned. As one can see, the structure of going through the work items and originators is similar to the push allocation.

Definition 56 (Maximal like factor push). *Let W be a list of work items, let R be a list of originators, then the maximal like factor for push allocation is define as:*

$$
\begin{aligned}
maxLikeFactorPush(W, R) &= maxLikeFactorPush(W, R, R, 0.0) \\
maxLikeFactorPush(w :: W, r :: R, R', d) &= \\
&\max(L_f(w, r), maxLikeFactorPush(w :: W, R, R', \max(L_f(w, r), d)) \\
maxLikeFactorPush([], R, R', d) &= d \\
maxLikeFactorPush(w :: W, [], R', d) &= \\
&maxLikeFactorPush(W, R', R', d) \text{ if } d = 0.0 \\
maxLikeFactorPush(w :: W, [], R', d) &= d \text{ if } d > 0.0
\end{aligned}
$$

For pull allocation, we have a similar computation for the maximal like factor as in the push allocation case. The main difference is in the order of traversing the work items and originators.

Definition 57 (Maximal like factor push). *Let W be a list of work items, let R be a list of originators, then the maximal like factor for push allocation is define as:*

$$maxLikeFactorPull(W, R) = maxLikeFactorPull(W, R, W, 0.0)$$
$$maxLikeFactorPull(w :: W, r :: R, W', d) =$$
$$\max(L_f(w, r), maxLikeFactorPull(W, r :: R, W', \max(L_f(w, r), d)))$$
$$maxLikeFactorPull(W, [], W', d) = d$$
$$maxLikeFactorPull([], r :: R, W', d) =$$
$$maxLikeFactorPull(W', R, W', d) \text{ if } d = 0.0$$
$$maxLikeFactorPull([], r :: R, W', d) = d \text{ if } d > 0.0$$

The best is determined with the previously defined like factor. In our current implementation, we have a like factor of 1.0 if an originator may execute a work item, and 0.0 otherwise. Using the aforementioned properties of an originator, we can take other aspects into account in determining the like factor without the need to change any of the perspectives other than the functions for the core controller.

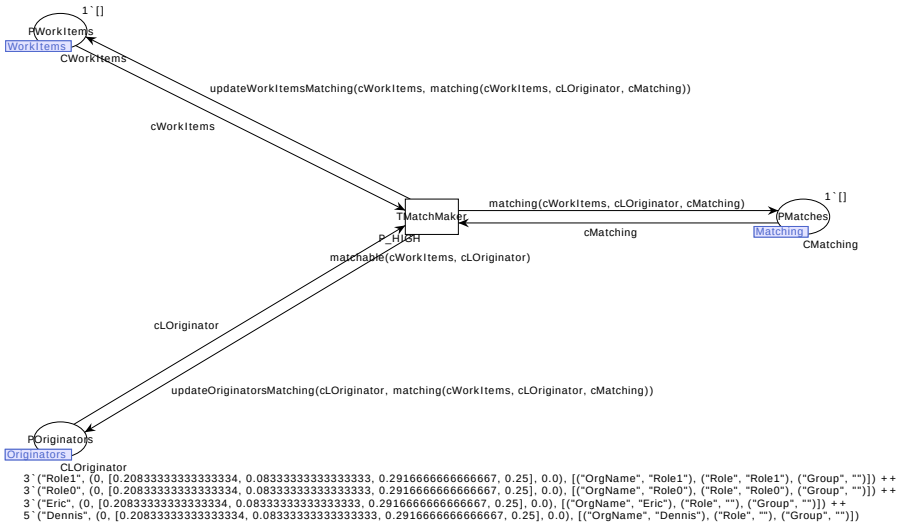

Fig. 21: The core controller responsible for the allocation of work items to originators.

7 Implementation

The implementation of *Petra* consists of the work done on the lefthand side of the bar with A and B in Fig. 22. On the righthand side of the bar, we have brought the first tool into *Petra* namely CPN Tools (Fig. 23). Both the lefthand side and righthand side are treated in isolation in the remainder of this section.

7.1 The framework

Petra is implemented as a ProM 6 package[7] called *Petra*. This means that our implementation can be used on user-created models and on mined models. The input to *Petra* is a Process Tree.

Within *Petra*, we iterate through the different models obtainable from the Process Tree (this may be just a single one). First, we construct a new configuration and apply this on the configurable model and obtain M (Fig. 22). This model is offered on interface A to a tool, and, on interface B, we receive a model enriched with the properties obtainable from the used tool. In case of CPN Tools, we obtain a model enriched with sojourn, processing, queue, and wait times. Finally, we add this model to the set of analysed models.

To guarantee the same configuration is not generated twice, we represent a configuration as a sequence of bits. Basically, each configuration option is encoded as a single bit, i.e., selected or not selected. We now only need to store a single bit sequence to know which configurations have been analysed and which have not been analysed. This in contrary to a naive implementation using a recursive function to iterate through the different configurations. Furthermore, our approach allows for a distributed computation on ranges of bit sequences, and the possibility to start the iteration from a given configuration (bit sequence).

As there is no control on the properties a particular perspective may contain, we have annotated every property to denote to which perspective(s) it belongs. This to prevent accidental modification due to reading a perspective not adhering to the subdivision of properties amongst perspectives. Apart from having an annotation, properties also have an identifier associated with them to differentiate similar properties with different semantics, e.g., costs can have a different meaning in different organisations, but both organisation would still like to call them costs.

For allowing a user-friendly experience, and the possibility to modify/create properties, every property registers a GUI with which the value of the property can be rendered and modified. Finally, to allow for storage of the process model with properties, every property has to implement read and write functionality of the property with values. This read a write is based on XML, and each may specify its own subschedule as this is handled completely by the property itself again.

The interfaces and the tools are implemented as a single programming interface. Every tool needs to provide an implementation of the different functions

[7] ProM can be downloaded from www.processmining.org

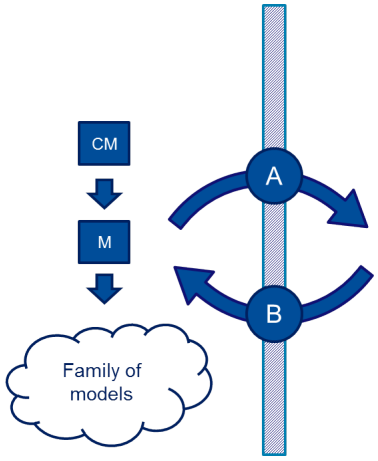

Fig. 22: Implementation of *Petra*

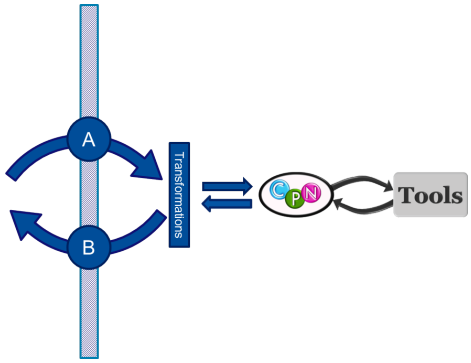

Fig. 23: Implementation of CPN Tools in *Petra*

required. The tool needs to implement a function for computing the dependent properties. Furthermore, the tool should register which dependent properties it can compute, and which (in)dependent properties it requires for doing so. To allow for future extensions of the tool interface, we have included an abstract tool interface at the base of every tool. If the tool interface changes the abstract tool interface can accommodate for missing functionality of older tool implementations.

7.2 CPN Tools

The implementation of bringing CPN Tools inside our framework entails three steps: transforming a Process Tree to a CPN model, running the simulator on the generated CPN model, and parsing the output of the simulator to enrich the

Process Tree. The transformation is discussed in Sect. 6. Here, we elaborate on running the simulator and parsing the output of the simulator.

Access/CPN, the simulator: We have made a transformation from a Process Tree to a CPN model to be used in CPN Tools. In this CPN model, we measure the sojourn time, i.e., the total time a case spent in a (sub)process, the queue time, i.e., the time a work item waited on a resource, the wait time, i.e., the time a case spent waiting to be synchronised, and the processing time, i.e., the actual work resources have performed on the activities. One of the advantages of CPN Tools is the existence of Access/CPN [19]. Access/CPN provides an interface to run the simulator without any human involvement. This is an important feature as the amount of possible instantiations for a Process Tree can be exponential in the number of configuration options.

Parsing the output: As mentioned, we measure the sojourn time, queue time, wait time, and processing time. These values are written in different files as tuples of case identifier, node identifiers, and the actual value. When parsing the output, every node is enriched with the aggregations of different values measured in the different replications run by the simulator.

Having presented all information about the framework, we now use *Petra* in a case study.

8 Case study

To demonstrate the use of *Petra*, we have taken the building permit process of two Dutch municipalities participating in the CoSeLoG project. Instead of simulating the entire building permit process, we focus in particular on the *Beroep en Bezwaar* (*objection and appeal*) subprocess. The *Beroep en Bezwaar* subprocess is to allow people to object and appeal to the disposal of a building permit. Conferment means that a building permit is granted to the requester. This subprocess starts after the municipality has published the disposal of a building permit. The disposal of a building permit means that the municipality agrees with the building permit but it is not yet definitive, i.e., based on objections and appeals the municipality may decide to disagree with the building permit.

The objective for carrying out this case study is twofold: (1) To demonstrate that the results obtained by *Petra* closely resemble reality, and (2) to show that *Petra* can indeed *automatically* analyse a family of process models. From the municipality event logs, we obtained the different perspectives for each of the municipalities. These perspectives have been combined into a Process Tree. In order to verify that the perspectives were obtained correctly, we first try to reproduce the behaviour recorded in the original logs from the municipalities. This way we can verify that the perspectives are encoded correctly.

8.1 Verifying perspectives

The characteristics of the logs are listed in Table 1. Furthermore, the logs span a time period of roughly 2 years. The logs consist of *create* events and of *completed*

Characteristics	Municipality A	Municipality B
Number of cases	302	481
Number of event	586	845
Number of event classes	15	23
Number of originators	5	4

Table 1: The characteristics of the logs used in the case study

events, and every event has an executor associated with it. The occurrences of the different event classes varies significantly. The least occurring event class occurs just once, while the most occurring event class occurs 262 times for Municipality A and 451 times for Municipality B. We only estimate processing times of activities with at least 3 observations in the log, this to have somewhat reliable values for the sojourn time of events whilst not disposing too many activities.

From the event logs, we have constructed a Process Tree only consisting of the control-flow perspective. In order to encode the probabilities for choosing a particular branch in the Process Tree, we have introduced variables which with a certain probability are "0" or "1" denoting true and false respectively. Since our choice constructs are binary, true indicates the first branch is executed and false indicates the second branch is executed.

Apart from the flow of the different cases, we also need to estimate the resource behaviour. First, we estimated the work schedule of the resources. The work schedule is estimated based on observations from the log. We have taken the timestamps of all events and we made the following assumptions:

- People work in shifts
- There are two shifts: A morning shift and an afternoon shift
- As there is no clear signal of breaks, we assume the shifts are consecutive
- If we have at least one measurement in a shift, we assume that this person is available for the process during that whole shift
- There is a weekly iteration, e.g., if someone worked on a Monday, we assume she can work every Monday

Based on these observations, we obtain the schedule as shown in Fig. 24. The top half of the schedule is for Municipality A, the bottom half is for Municipality B.

For the sojourn times of the activities, we have taken three separate approaches and compared the results in order to have reliable estimations. The first approach is based on comparing completion times between non-parallel consecutive events. The second approach is based on aligning the log to the earlier obtained control-flow perspective using the work in [37]. Finally, the third approach is based on observing the creation time of a work item in comparison to the completion time of that work item.

The first and second approach, yield results similar to each other, i.e., there is no significant difference. The third approach yielded negative values indicating these values are less reliable. Apart from taking the values from the log, we also

		Monday		Tuesday		Wednesday		Thursday		Friday	
A	560519										
	560521										
	560532										
	560458										
	4634935										
B	560429										
	560602										
	560602										
	1254625										

Fig. 24: The work schedule used, the top work schedule is for Municipality A, the bottom is for Municipality B.

consulted legislation related to the process at hand. This legislation specified legal time intervals, e.g., after publishing the disposal of a building permit, there is a 6 weeks time interval in which people may object to the conferment. These time intervals are less visible in the log as there is a large variation, e.g., on average 6.6 weeks, with a standard deviation of 2 weeks. When a person objects to a conferment, there is a legal limit within which the municipality has to respond. However, the law makes an exception for complicated cases. As the information was not present in the log about which cases have been classified as complicated, we can only try to make an educated guess on the sojourn time of such an activity.

To take the legal constraints into account, we have explicitly added activities to model these time intervals. Where the law is unclear, we estimated the time interval in a number of weeks, as all the times observed are specified in terms of weeks. For the activities in which the law allows an exception with respect to the legal time interval, we have made an educated guess based on the activity description. For instance, the activity "Beroep ingesteld" (*Appeal rated*) is assumed to take on average 30 minutes as one has to read the appeal and make a judgement based on that.

Next to the legal time intervals, we also have taken into account the variability of the working speed of the different resources, i.e., the sojourn time of different resources executing the same activity may vary. However, for most activities there were no significant difference in the sojourn times. Therefore, we estimate the sojourn time of these activities on the complete set of values. Furthermore, we have abstracted from outliers.

Since the log consists of event codes, we have included Table 2. This table converts the event codes to activity names, which makes it easier to understand the amount of work required for each step. The estimated values for the different sojourn times are listed in Table 3.

Event code	Event Name
540	Objection to disposal submitted
550	Treat objection
550_1	
560	Objection wrapped up
590	Received request for preliminary verdict
600	Treat preliminary verdict
610	Preliminary verdict wrapped up
630	Appeal set
640	Received request for preliminary verdict
670	Treat appeal
680	Appeal wrapped up
700	
730	Contested disposal affected
740	Verdict given by court
760	
765	
766	
770	Establish decision phase original decree
775	
780_1	Create decree for the purpose of the disposal of the court
780_2	Connect disposal court
780_3	Register date of disposal of court
790	Establish decision phase of the verdict of court

Table 2: The event codes and the description (if given).

Next to the work schedule and timing information, we also need to know which originators are allowed to execute which activity. This information is summarised in Table 7. The left side of the table is for Municipality A, and the right side is for Municipality B. Using this information, we were able to construct the simulation models for both Municipality A and Municipality B. We used the following simulation property in the experiment perspective:

- The queue principle is FiFo.
- We use push allocation
- We have a warmup period of 150,000 steps in the simulator
- The number of replications is 30
- The replication length is 150,000 steps in the simulator

The results per replication are listed in Table 8 and Table 9. The average sojourn time for Municipality A is 9.5 weeks (1588.49 hours) with a standard deviation of 1 day (23.66 hours). For Municipality B, the average sojourn time is 5 weeks (828.34 hours) with a standard deviation of 14.25 hours.

Since the log can be seen as a single long replication, we used batch means to be able to compute replications and to be able to compare the log to the simulation results. See [38] for an overview of the batch means method. Since our

Event code	#		Time out (weeks)		Avg. Proc. Time (min.)	
	A	B	A	B	A	B
540	262	451	10	1 + 10 `beta(5.51, 7.88)`	`norm(10, 3)`	`norm(10, 3)`
550	3	8	0	4	`norm(1, 0.3)`	`norm(30, 9)`
550_1	1					
560	2	6		0		0.23, `norm(0.08, 0.00)`
590	2	8		5		`norm(10, 3)`
600		3		4		`norm(30, 9)`
610		3		0		0.1, 0.03
630	29	31	6	5	`norm(30, 9)`	`norm(30, 9)`
640		3		3		`norm(10, 3)`
670	1	2				
680	1	2				
700		2				
730	17	26	0.57	0	`norm(10, 3)`	0
740		1				
755	17	4		0	822.12, 0.52, `norm(0.33, 0.05)`, `norm(0.45, 0.02)`	`norm(0.125, 0.00)`, 0.07
760		1				
765	62	110		0	`norm(0.00, 0.00)`	`norm(0.16, 0.00)`
766		3		0		0.22
770	187	170	10	4	`norm(10, 3)`	`norm(0.22, 0.03)`
775	2	2				
780_1		1				
780_2		1				
780_3		1				
790		1				

Table 3: The activities with the time outs and the processing times for Municipality A and Municipality B.

dataset is small, we cannot have too many batches, i.e., then there are batches which do not contain any observations, but if we have too few batches then the 95% confidence interval become too large. Furthermore, the batches cannot become too small, because then the batches are correlated. For Municipality A, we have 16 batches and for Municipality B we have 25 batches. Fig. 25 shows the results, where Municipality A is shown left and Municipality B is shown right and times on the y-axis are in hours.

As one can clearly see, there is overlap in the box plots of the logs and the simulations. Hence, we cannot conclude that our simulation model and reality differ in an unacceptable manner.

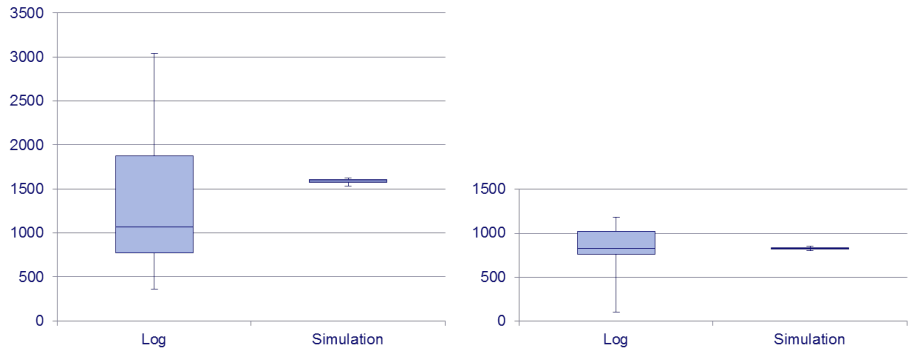

Fig. 25: The box plots of the logs and the results of simulating the individual CPN models, left is Municipality A and right is Municipality B. The y-axis is in hours.

8.2 Analysing a family of process models

At this point, we have shown that it is possible to closely mimic the execution of business processes with *Petra*. We now combine the models from Municipality A and Municipality B into a single Process Tree. This Process Tree is the union of Municipality A and Municipality B but allows for more than the behaviour displayed by Municipality A and Municipality B. Now let us suppose that we want to use this combined Process Tree to seek for ways to improve the sojourn time for Municipality A by potentially changing its process, borrowing behaviour from Municipality B. Since not all combinations make sense, e.g., Municipality A will not hire the employees of Municipality B, we limit the configurability of the Process Tree. This means that we take the characteristics of the employees of Municipality B into account, e.g., their work speed, but do not put the employees of Municipality B into the Process Tree.

For the case study, we mainly focus on the activities *540 Objection to disposal submitted*, *630 Appeal set*, and *770 Establish decision phase original decree* since these activities embody part of the significant difference between Municipality A and Municipality B. Furthermore, since the building permit process is heavily subjected to legal requirements, there is not so much room to change what is done but there are opportunities to change how things are done.

With the focus on the activities *540*, *630*, and *770*, we have a solution space of 8 models (all combination for performing an activity either according to Municipality A or Municipality B). After simulating all of these models, we obtain the results (sojourn times of the entire process in hours) as in Table 4 (the full set of results is in Table 10).

As one can see, working on the activities *540* and *770* in the same way as Municipality B would indeed significantly decrease the sojourn time for Municipality A. Furthermore, one can clearly see that changing *770* and *630* without

Table 4: The different work speeds for the activities and the effects on the simulation (note that 2 is the original Municipality A Process Tree).

	1	2	3	4	5	6	7	8
540	A	A	A	B	A	B	B	B
630	A	A	B	A	B	A	B	B
770	B	A	A	B	B	A	B	A
Avg	1488.78	1572.48	1570.91	806.60	1496.11	1217.32	803.67	1224.50
Std. dev.	29.55	33.82	18.58	22.93	24.81	34.80	20.24	33.10

changing *540* will not yield any significant improvements. Finally, *630* does not have any significant impact on the performance of the process.

By enabling the highly realistic simulation of actual business processes and supporting the search for altering a process in search for performance improvement, we have shown the feasibility and potential of *Petra*.

9 Conclusion and future work

We have presented *Petra*, a generic and extensible toolset for the analysis of a family of process models. The genericity is achieved by not limiting our set of performance indicators or the set of properties a process model may contain. The extensibility is achieved by allowing any analysis tool to be used. Finally, the implementation of our framework has been used to conduct a case study that involves the data from two municipalities cooperating in the CoSeLoG project. In particular, we have demonstrated how potential improvements of an existing process can be established.

Our framework can be extended in a range of directions to make it even better equipped for the task at hand, but also for future tasks. In the remainder of this section, we touch on some of these.

Currently, the simulation of each of the process models is a time-consuming task. In the experiments, simulating a single model took around 3 days on a single core of 2.80 GHz. Since simulation takes such a long time, one would like to minimise the amount of models to be simulated. In our current implementation, we naively iterate through all possible instantiations of the configurable process model. This means that equivalent models, obtained by different configurations, are analysed multiple times. Defining equivalence classes on configurations and taking these into account in the iteration through the models could already result in a significant speed up. Another approach to minimise the number of to-be-analysed models, is to gather knowledge beforehand on the performance measures a user wants to optimise. In this way, we would lose some of the reusability of the results but it would be possible to obtain quick estimates on the relevant measures to decide if simulating the process model will be worthwhile.

Related to quick estimations of performance measures, we also want to cater for different analysis tools in our framework. This would make it possible to use work from simulation and queueing theory to obtain timing information.

Enabling us to chain these tools starting from fast and imprecise to slow and precise. If different tools for the same performance measure are used, one might want to maintain the different measures computed by the different tools. Currently the assumption is that the different tools give the correct value. Hence, there is no need to doubt the values returned by the last tool used for a property. However, it might be beneficial to remember a history of computed values (from rough estimations, to exact value). This to be able to compare rough estimations of one model with another, but also to be able to show the discrepancies between results obtained from different analysis tools.

In our current implementation, we have limited ourselves to the time perspective. The devil's quadrangle [20], contains 4 different directions, i.e., time, cost, flexibility, and quality (internal and external). We want to extend our framework to also incorporate the other dimensions. Furthermore, to be able to compute these new dimensions, we want to included a larger variety of tools, and be able to automatically chain tools to obtain the desired dimensions.

Currently, the user obtains a (large) set of analysed models. We want to support the user in navigating through this set to be able to find her desired model quicker. This navigation could happen with the use of a Pareto front. A Pareto front is basically a method of finding the (set of) optimal solution(s) in n-dimensional space [39]. In this n-dimensional space, every property is a dimension. By offering an interface for the user to specify the desired solution direction, we can support the user maximally in obtainment of the best process models from which the user can select the most desired model based on her expert knowledge.

Finally, we want to support the user in migrating from a given configurable process model, a configuration and a set of cases to the new improved process model found by *Petra*. By supporting this, we would be able to support processes where cases have a long lifetime as well as supporting escalations of the process model [40], i.e., changing the behaviour of the process model at run-time. Related to the escalations, we would also like to be able to encode the current state of the workflow management system to be able to predict the near future and the short-term effects of different configurations.

A Conversion to CPN Tools

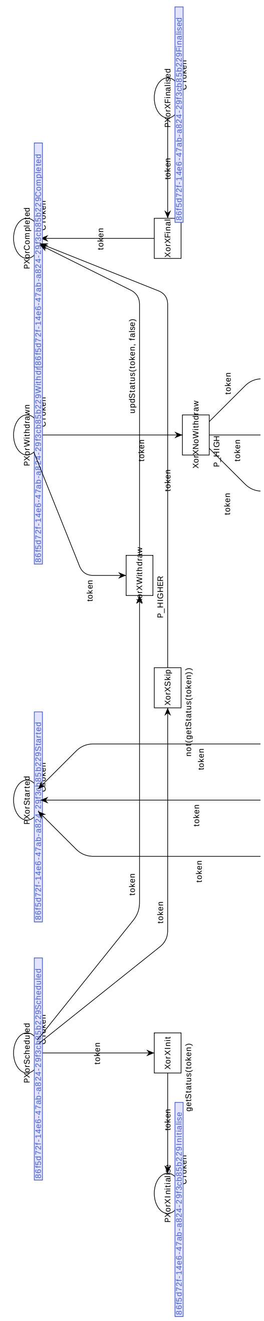

Fig. 26: The interface a node (XOR in this case) has towards its parent.

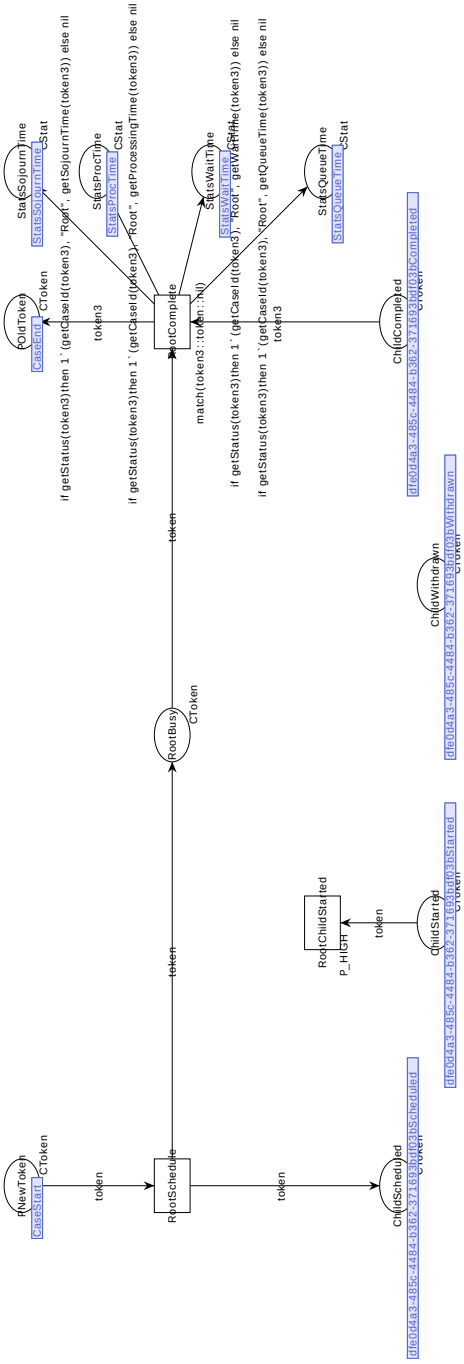

Fig. 27: The root net connecting the arrival of cases with the control flow.

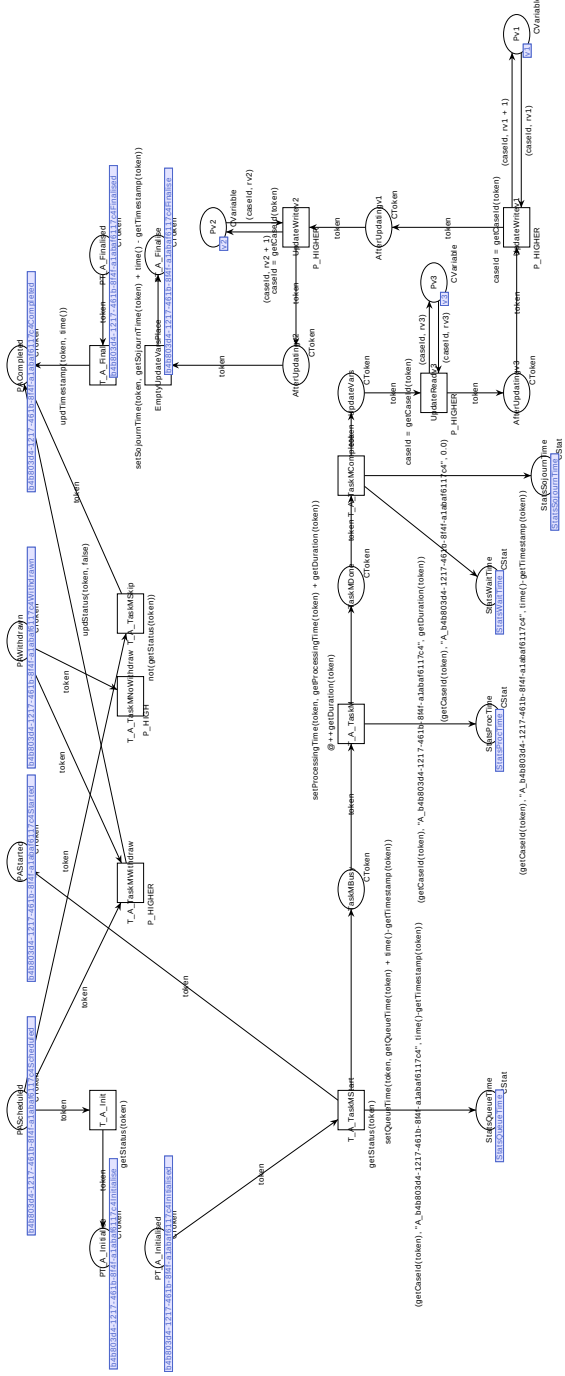

Fig. 28: Manual task encoding.

Fig. 29: Task controller encoding.

Fig. 30: SEQ operator encoding.

Fig. 31: AND operator encoding.

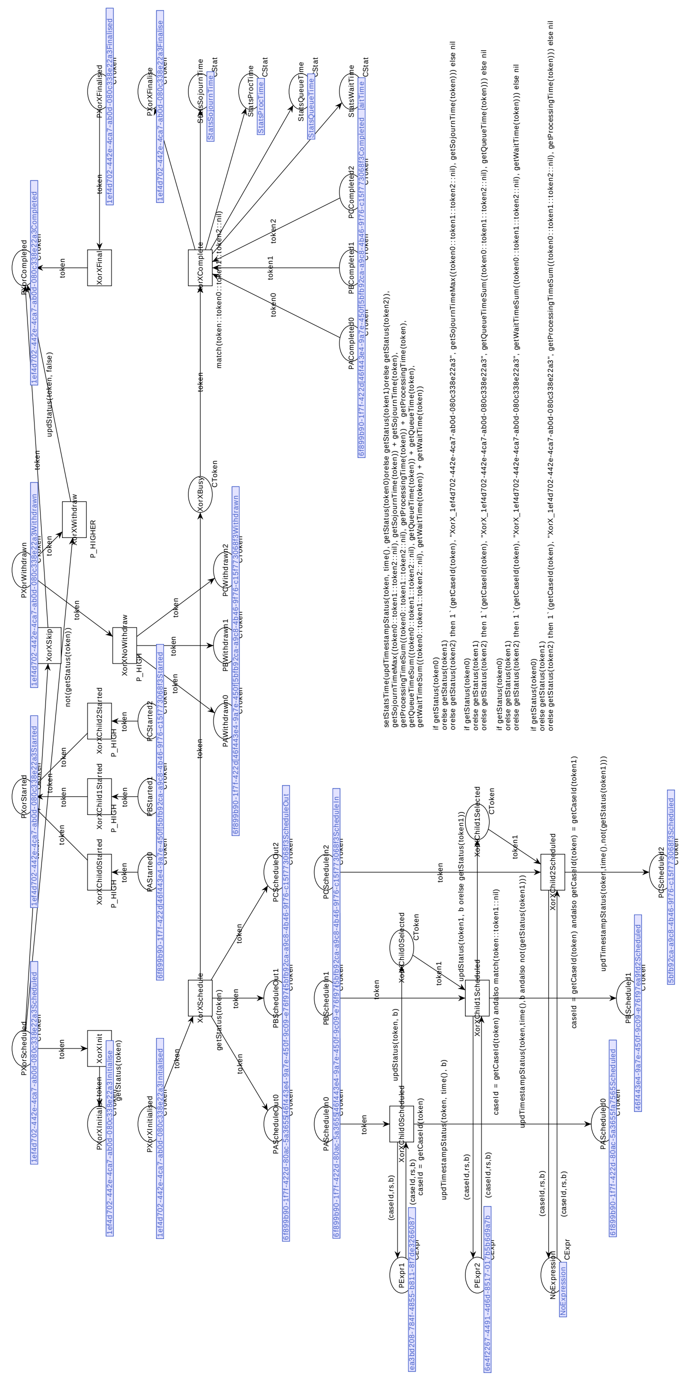

Fig. 32: XOR operator encoding.

Fig. 33: LOOPXOR operator encoding.

Fig. 34: OR operator encoding.

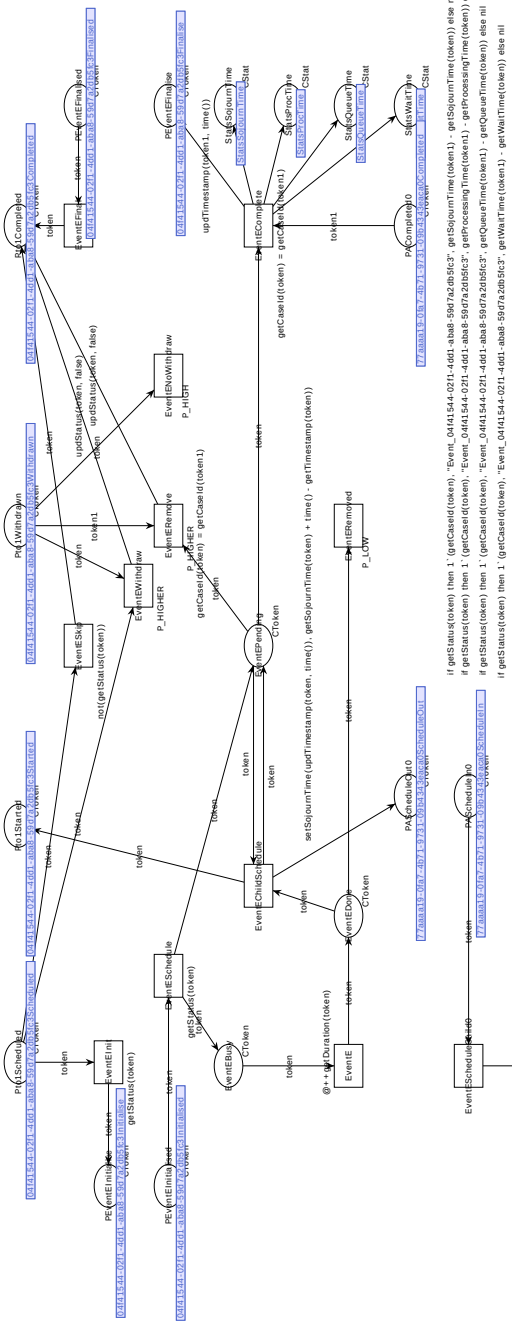

Fig. 35: EVENT operator encoding.

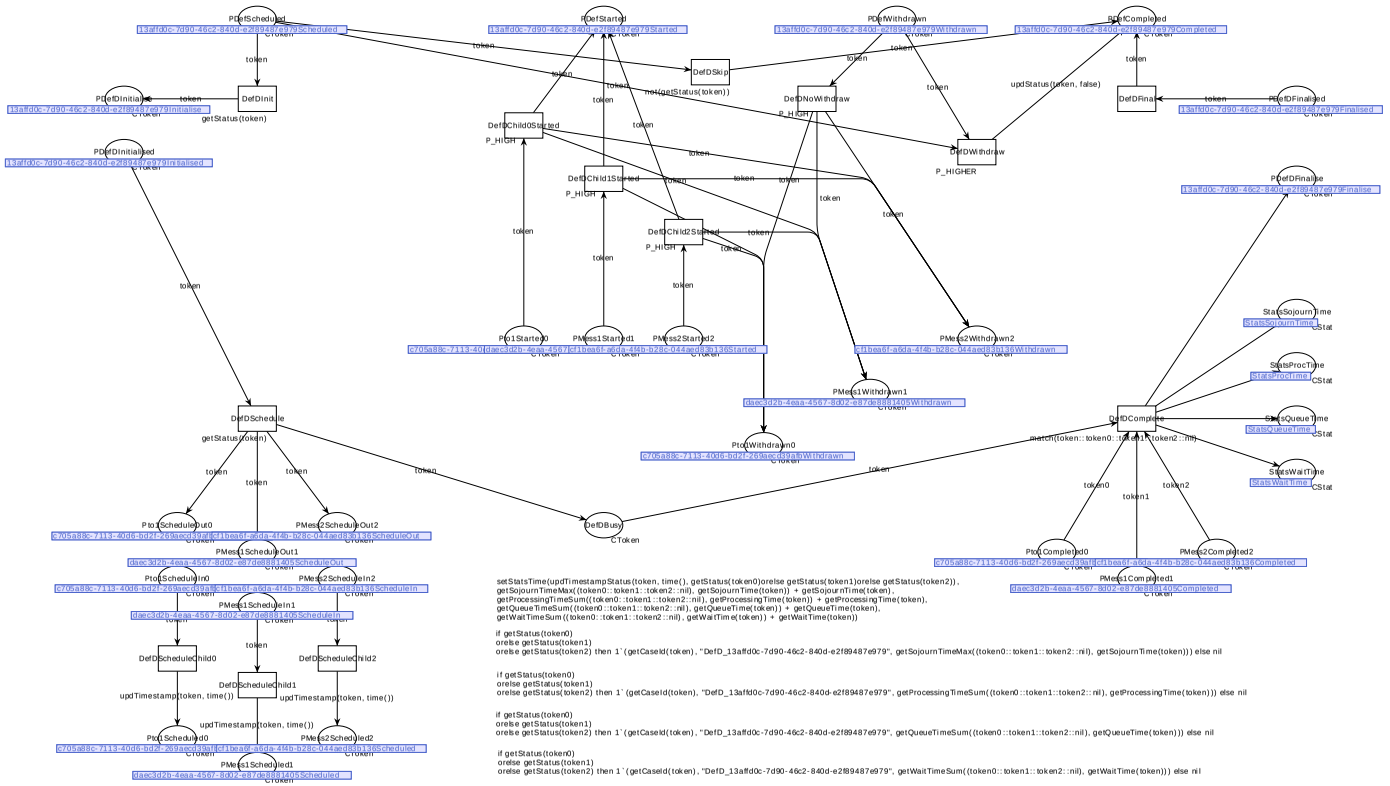

Fig. 36: DEF operator encoding.

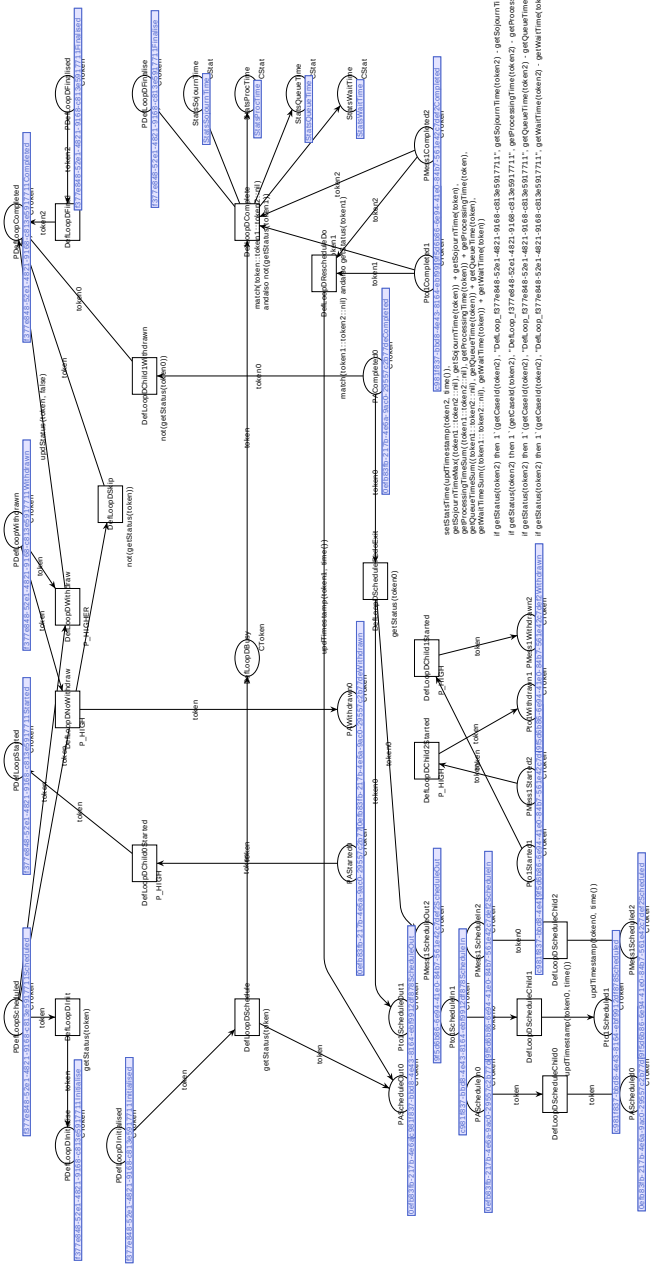

Fig. 37: LOOPDEF operator encoding.

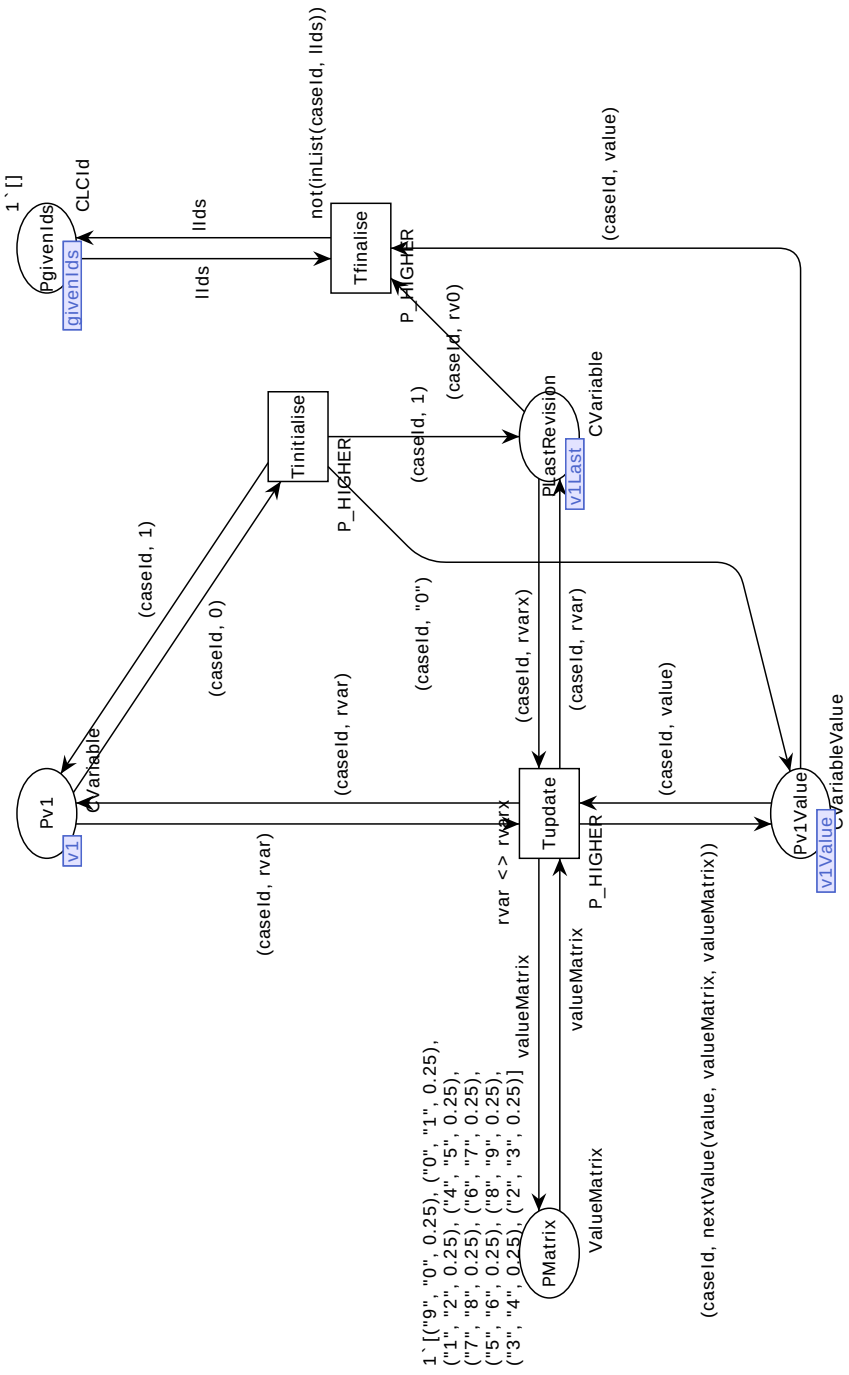

Fig. 38: Variable encoding.

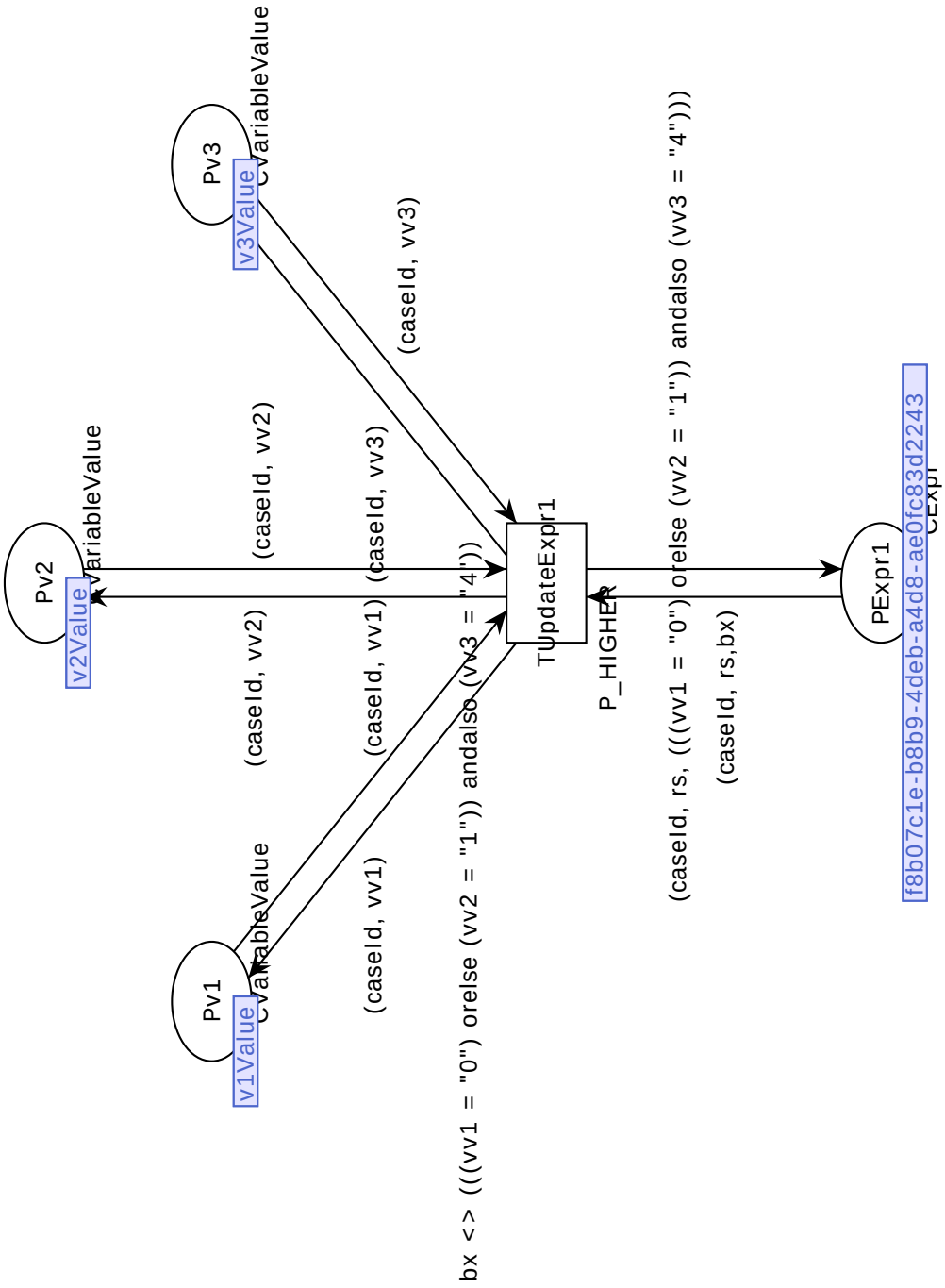

Fig. 39: Expression encoding

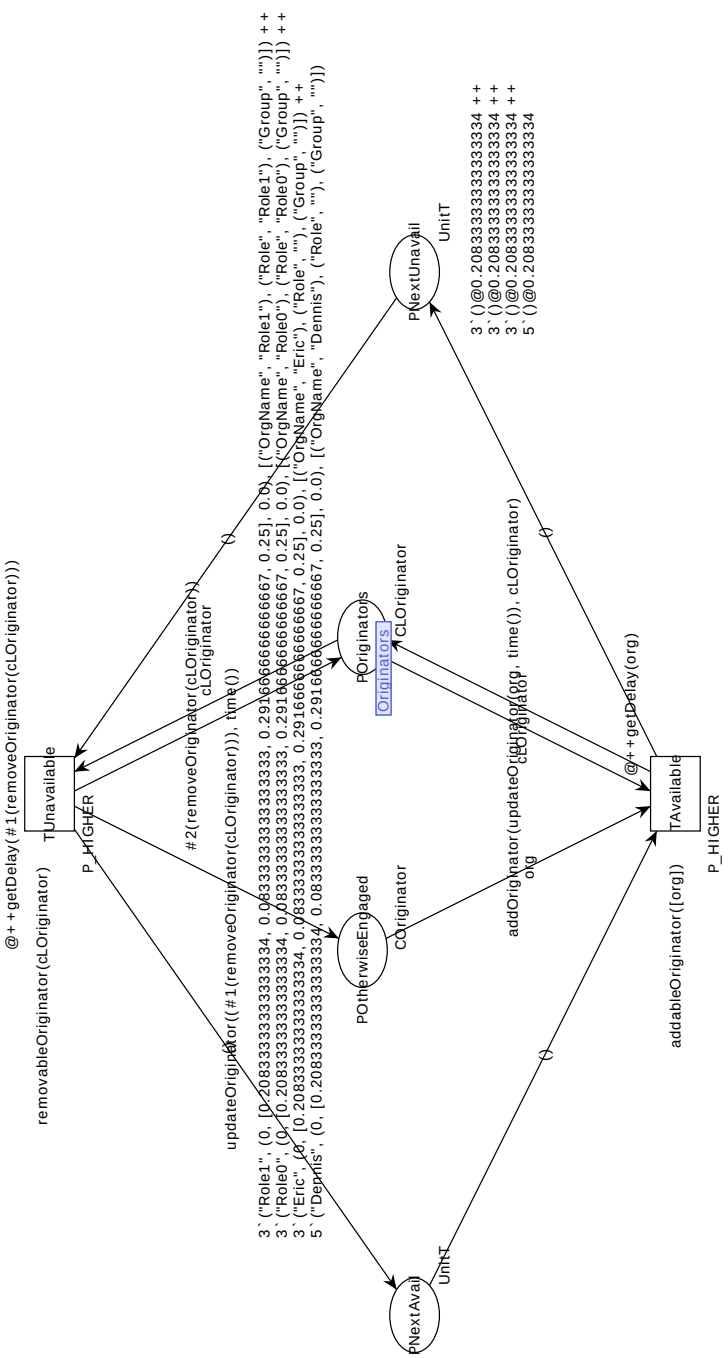

Fig. 40: The work schedule for the different originators.

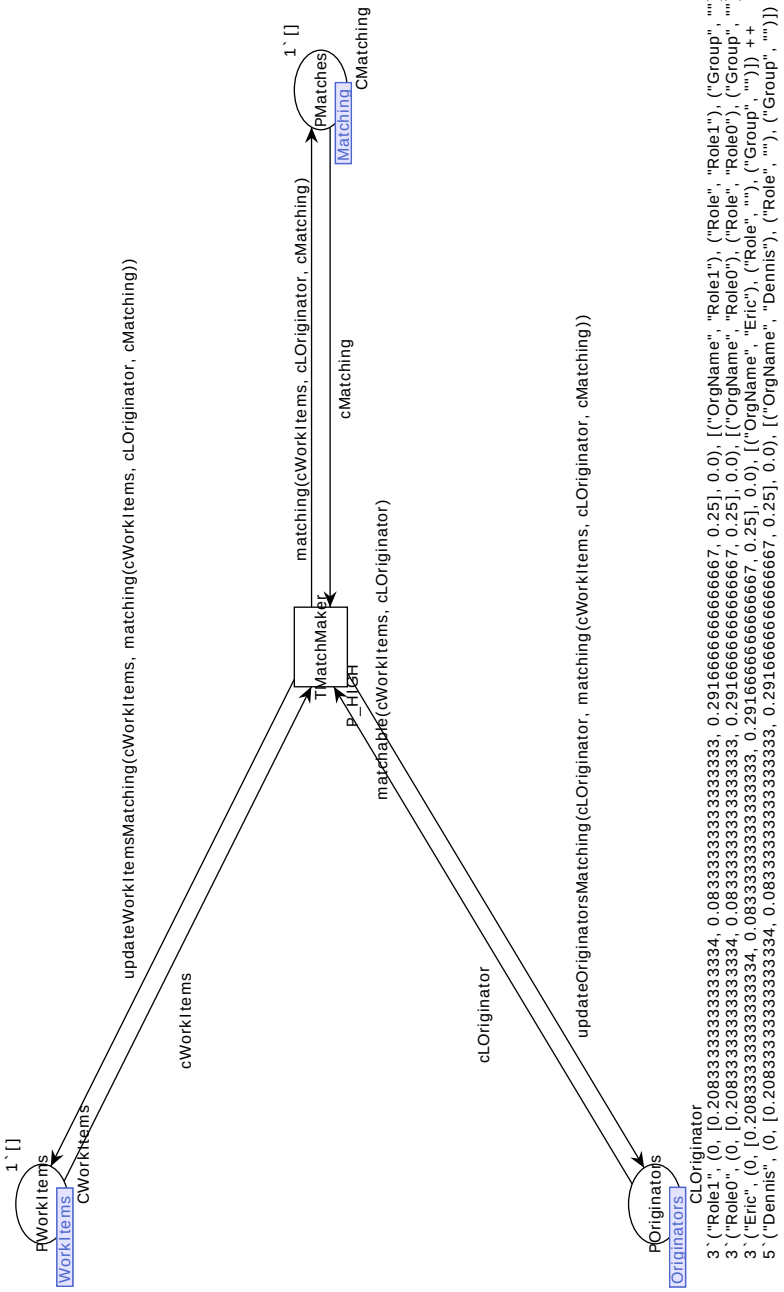

Fig. 41: The core controller responsible for the allocation of work items to originators.

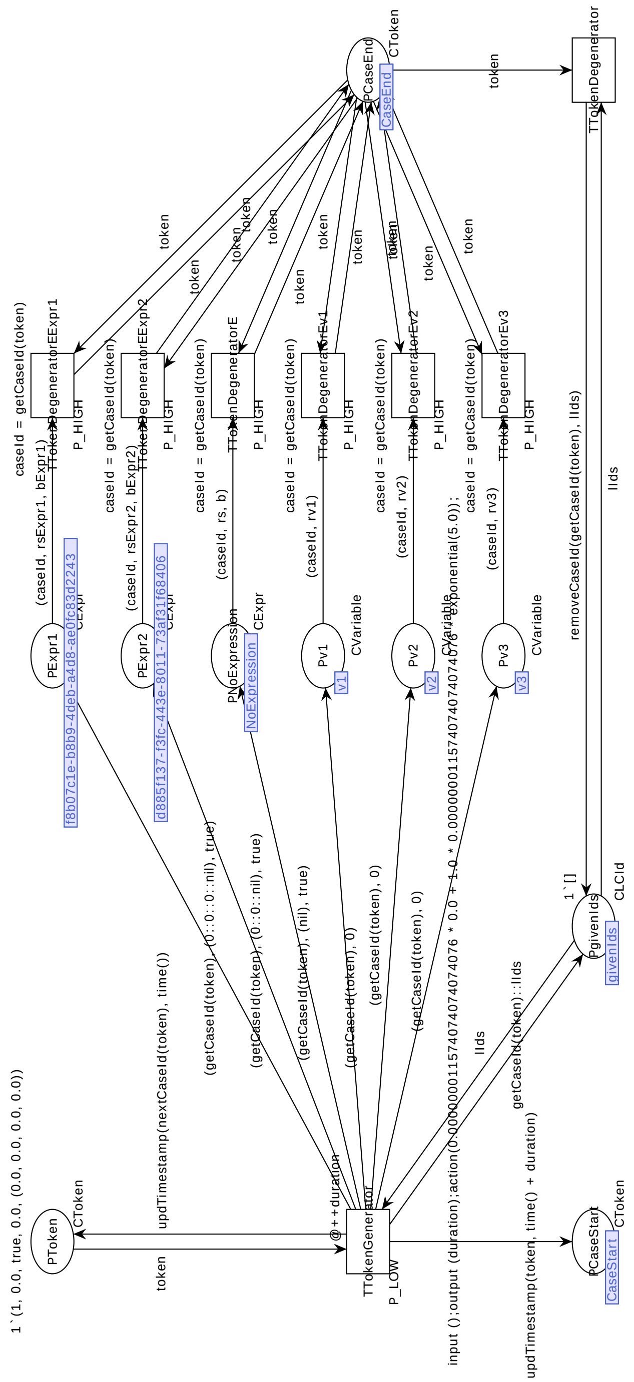

Fig. 42: The arrival controller responsible for the arrival process of cases.

B CPN Types and Functions

Here we have listed all of the CPN Types and CPN Functions present in the transformation from a Process Tree to a CPN model. Note that not all functions are currently used but are present due to historical reasons.

B.1 CPN types

In CPN Tools, we have the following colsets:

Type name	Type specification
CTime	real
CTimes	list CTime
CDuration	real
CId	int
CDpe	bool
CSojourn	real
CProcessing	real
CQueue	real
CWait	real
CPerformance	product CSojourn * CProcessing * CQueue * CWait
CToken	product CId * CTime * CDpe * CDuration * CPerformance timed
CTokens	list CToken
CLabel	string
CStat	product CId * CLabel * CTime
CRevision	int
CRevisions	list CRevision
CVariable	product CId * CRevision
CExpr	product CId * CRevisions * BOOL
CLCId	list CId
Key	string
Value	string
CVariableValue	product CId * Value
Property	product Key * Value
Attributes	list Property
Avail	list REAL
Schedule	product INT * Avail * CTime
COriginator	product STRING * Schedule * Attributes
COriginatorCId	product CId * COriginator
CWorkItem	product CToken * Attributes
CWorkItems	list CWorkItem
CLOriginator	list COriginator
CMatch	product CWorkItem * COriginator
CMatching	list CMatch
ValueVector	product Value * Value * REAL
ValueMatrix	list ValueVector
CLCIds	list CId
DomCod	product REAL * REAL
YerkesD	list DomCod
UnitT	unit timed

Table 6: The colsets used

B.2 CPN Functions

Constants We have the following constants for determining the priority of a transition:

```
val P_LOW=10000;
val P_HIGH=100;
val P_HIGHER=10;
```

max The max function gives the maximum of a list of time values.

```
fun max (nil:CTimes) = 0.0 | max (head::tail) = if head >
    max(tail) then head else max(tail);
```

getCaseId Gives the case identifier of a token.

```
fun getCaseId (token:CToken) = #1 token;
```

getTimestamp Gives the timestamp of a token.

```
fun getTimestamp (token:CToken) = #2 token;
```

getStatus Gives the status of a token.

```
fun getStatus (token:CToken) = #3 token;
```

getDuration Gives the duration of a token, this duration is used in tasks.

```
fun getDuration (token:CToken) = #4 token;
```

updTimestamp Updates the timestamp of a token.

```
fun updTimestamp (token:CToken,timestamp:CTime) = (#1
   token, timestamp, #3 token, #4 token, #5 token);
```

updStatus Updates the status of a token.

```
fun updStatus (token:CToken, status:CDpe) = (#1 token, #2
    token, status, #4 token, #5 token);
```

updTimestampStatus Updates the timestamp and status of a token.

```
fun updTimestampStatus (token:CToken, timestamp:CTime,
   status:CDpe) = (#1 token, timestamp, status, #4 token,
    #5 token);
```

updDuration Updates the duration of a token.

```
fun updDuration(token: CToken, duration: CDuration) = (#1
    token, #2 token, #3 token, duration, #5 token);
```

match Checks whether a list of tokens all have the same identifier.

```
fun match (nil:CTokens) = true | match (token::nil) =
    true | match (token1::token2::tokens) = (getCaseId(
    token1) = getCaseId(token2)) andalso match(token2::
    tokens);
```

write This function is used to write the performance statistics to a file.

```
fun write((cId, label, cTime) :: cStat) = Int.toString(
    cId) ^", " ^ label ^", " ^ ModelTime.toString(cTime) ^
     "\n" ^ write(cStat) | write(nil) = "";
```

probability Gives a probability from a bernoulli experiment given a probability.

```
fun probability(p: real) = if bernoulli(p) = 1 then true
    else false;
```

getSojournTime Gives the sojourn time of a token.

```
fun getSojournTime(token: CToken) = #1 (#5 token);
```

getProcessingTime Gives the processing time of a token.

```
fun getProcessingTime(token: CToken) = #2 (#5 token);
```

getQueueTime Gives the queue time of a token.

```
fun getQueueTime(token: CToken) = #3 (#5 token);
```

getWaitTime Gives the wait time of a token.

```
fun getWaitTime(token: CToken) = #4 (#5 token);
```

setSojournTime Sets the sojourn time of a token.

```
fun setSojournTime(token: CToken, timestamp: CTime) = (#1
    token, #2 token, #3 token, #4 token, (timestamp,
    #2(#5 token), #3(#5 token), #4(#5 token)));
```

setProcessingTime Sets the processing time of a token.

```
fun setProcessingTime(token: CToken, timestamp: CTime) =
    (#1 token, #2 token, #3 token, #4 token, (#1(#5 token)
    , timestamp, #3(#5 token), #4(#5 token)));
```

setQueueTime Sets the queue time of a token.

```
fun setQueueTime(token: CToken, timestamp: CTime) = (#1
    token, #2 token, #3 token, #4 token, (#1(#5 token),
    #2(#5 token), timestamp, #4(#5 token)));
```

setWaitTime Sets the wait time of a token.

```
fun setWaitTime(token: CToken, timestamp: CTime) = (#1
    token, #2 token, #3 token, #4 token, (#1(#5 token),
    #2(#5 token), #3(#5 token), timestamp));
```

setStatsTime Sets the sojourn, processing, queue, and wait time of a token.

```
fun setStatsTime(token: CToken, timestampSojourn: CTime,
    timestampProcessing: CTime, timestampQueue: CTime,
    timestampWait: CTime) = (#1 token, #2 token, #3 token,
     #4 token, (timestampSojourn, timestampProcessing,
    timestampQueue, timestampWait));
```

getSojournTimeMax This function gets the maximal sojourn time in a set of tokens. We use an offset to be able to compensate for sojourn time already stored in the token but not relevant for this particular computation. For instance, when a part of the process model has already been executed, then the sojourn time is x, when we execute a subprocess then the sojourn time becomes $x + y$. In order to know the sojourn time for the subprocess, we subtract the offset (in this case x) if it is a true token.

```
fun getSojournTimeMax(nil: CTokens, offset: CTime) = Real
    .negInf | getSojournTimeMax(token::tokens: CTokens,
    offset: CTime) = if(getStatus(token)) then Real.max(
    getSojournTime(token) - offset, getSojournTimeMax(
    tokens, offset)) else getSojournTimeMax(tokens, offset
    );
```

getSojournTimeMin Similar to `getSojournTimeMax` but now yields the minimal sojourn time for a set of tokens.

```
fun getSojournTimeMin(nil: CTokens, offset: CTime) = Real
    .posInf | getSojournTimeMin(token::tokens: CTokens,
    offset: CTime) = if(getStatus(token)) then Real.min(
    getSojournTime(token) - offset, getSojournTimeMin(
    tokens, offset)) else getSojournTimeMin(tokens, offset
    );
```

getWaitTimeMax The `getWaitTimeMax` function computes the wait time at a synchronisation (AND, or OR). In order to be able to compute the wait time, we need to know the sojourn time of the of the entire subprocess. The difference between the sojourn time of the entire subprocess and the sojourn time of a particular child is the wait time. Note that, we included again the offset to compensate for sojourn time from a different subprocess.

```
fun getWaitTimeMax(nil: CTokens, sojournTime: CTime,
    offset: CTime) = 0.0 | getWaitTimeMax(token::tokens:
    CTokens, sojournTime: CTime, offset: CTime) = if(
    getStatus(token)) then Real.max(sojournTime -
    getSojournTime(token) - offset, getWaitTimeMax(tokens,
     sojournTime, offset)) else getWaitTimeMax(tokens,
    sojournTime, offset);
```

getProcessingTimeSum The `getProcessingTimeSum` computes the sum of the processing times of a list of tokens. The offset given to the function is removed from each of the processing times recorded by each of the tokens.

```
fun getProcessingTimeSum(nil: CTokens, offset: CTime) =
    0.0 | getProcessingTimeSum(token::tokens: CTokens,
    offset: CTime) = if(getStatus(token)) then (
    getProcessingTime(token) + getProcessingTimeSum(tokens
    , offset) - offset) else getProcessingTimeSum(tokens,
    offset);
```

getQueueTimeSum This function is similar to the `getProcessingTimeSum` apart from the fact it works with the queue time of this subprocess.

```
fun getQueueTimeSum(nil: CTokens, offset: CTime) = 0.0 |
    getQueueTimeSum(token::tokens: CTokens, offset: CTime)
     = if(getStatus(token)) then (getQueueTime(token) +
    getQueueTimeSum(tokens, offset) - offset) else
    getQueueTimeSum(tokens, offset);
```

getWaitTimeSum Similar to the `getQueueTimeSum` but now on the wait time.

```
fun getWaitTimeSum(nil: CTokens, offset: CTime) = 0.0 |
    getWaitTimeSum(token::tokens: CTokens, offset: CTime)
    = if(getStatus(token)) then (getWaitTime(token) +
    getWaitTimeSum(tokens, offset) - offset) else
    getWaitTimeSum(tokens, offset);
```

nextCaseId Creates a new token based on a given token, but increments the case identifier by one.

```
fun nextCaseId(token: CToken) = (#1(token) + 1, #2 token,
     #3 token, #4 token, #5 token);
```

getProperty Gives the value for a given key. Note that, keys and values are strings.

```
fun getProperty(k, nil) = "" | getProperty(k, (key, value
    )::xs) = if(k = key) then value else getProperty(k, xs
    );
```

addWorkItem Adds a work item to a list of work items, note that this function comes in different flavours dependent on the queueing principle selected. The functions are for FiFo, LiFo, SiRo. Note that, for the SiRo encoding, we include a function to insert the new work item in a random place in the list of work items.

```
fun addWorkItem(token: CWorkItem, xs: CWorkItems) = rev(
    token::rev(xs));

fun addWorkItem(token: CWorkItem, xs: CWorkItems) = token
    ::xs;

fun addWorkItemN(token: CWorkItem, nil, _) = [token] |
    addWorkItemN(token: CWorkItem, x::xs: CWorkItems, r:
    real) = if(r < 1.0) then token::x::xs else x::
    addWorkItemN(token, xs, r - 1.0);

fun addWorkItem(token: CWorkItem, xs: CWorkItems) =
    addWorkItemN(token, xs, uniform(0.0, Real.fromInt(List
    .length(xs))));
```

removeMatching Removes a matching from the list of matching given a work item.

```
fun removeMatching(token: CWorkItem, (a, b)::xs:
    CMatching) = if(a = token) then xs else (a,b)::
    removeMatching(token, xs) | removeMatching(token:
    CWorkItem, nil: CMatching) = [];
```

inMatching Determines if a work item is present in the list of matchings.

```
fun inMatching(w: CWorkItem, nil) = false | inMatching(w:
    CWorkItem, (w', r)::M: CMatching) = if(w = w') then
    true else inMatching(w, M);
```

getMatching Gives the matching belonging to a work item.

```
fun getMatching(w: CWorkItem, (w', r)::M:CMatching) = if(
    w = w') then (w', r) else getMatching(w, M);
```

addOriginator Adds an originator to the list of originators. This function is used when a task has completed, i.e., there are no originators created.

```
fun addOriginator(org, xs) = rev(org::rev(xs));
```

getLengthSchedule A function to determine the length of the schedule for an originator. This function was used for bounding the index such that it stays within the boundaries of the schedule.

```
fun getLengthSchedule(org: COriginator) = List.nth(#2 (#2
    org), (#1 (#2 org)));
```

removeElement Generic function to remove an element from a list of elements.

```
fun removeElement(a, nil) = nil | removeElement(a, b::xs)
    = if(a = b) then xs else b::removeElement(a, xs);
```

isRemovableOriginator This function determines for an originator if it can be removed (or added) to the process. First a check is done whether the schedule of an originator is empty, if this is the case, then the originator is always available to the process. If the schedule is not empty, then we check if the time from the previous change is larger than the time specified for this state. On the left hand side of the >=, we compute the time an originator has been (un)available to the process. The right hand side of the >= computes the time we are supposed to be (un)available to the process.

```
fun isRemovableOriginator(a: COriginator) = if(#2 (#2 a)
    = []) then false else if((time() - (#3(#2 a))) >= List
    .nth(#2 (#2 a), (#1 (#2 a)))) then true else false;
```

hasRemovableOriginator This function determines for a list of originators if there is an originator which can be made (un)available to the process.

```
fun hasRemovableOriginator(nil: CLOriginator) =  false |
    hasRemovableOriginator(a::xs: CLOriginator) = if(
    isRemovableOriginator(a)) then true else
    hasRemovableOriginator(xs);
```

getRemovableOriginator This function returns the removable originator form a list of originators. Note that, we use a default option, but this function may only be called if `hasRemovableOriginator` evaluates to true.

```
fun getRemovableOriginator(nil: CLOriginator) = ("", (0,
    [], 0.0), []) | getRemovableOriginator(a::xs:
    CLOriginator) = if(isRemovableOriginator(a)) then a
    else getRemovableOriginator(xs);
```

removeOriginator The first removable originator is removed from a list of originators, and returned in a tuple. If we have a list of originators xs and a removable originator o, then $(o, xs', true)$ is returned where xs' is the list of originators xs without o, and *true* signals success of the operation.

```
fun removeOriginator(nil: CLOriginator) = (("", (0, [],
    0.0), []), nil, false) | removeOriginator(xs:
    CLOriginator) = if(hasRemovableOriginator(xs)) then (
    getRemovableOriginator(xs), removeElement(
    getRemovableOriginator(xs), xs), true) else (("", (0,
    [], 0.0), []), nil, false);
```

removableOriginator This functions checks if there is a removable originator.

```
fun removableOriginator(xs: CLOriginator) = #3(
    removeOriginator(xs));
```

updateOriginator The `updateOriginator` function, updates the schedule of an originator. The `(((#1(#2 org)) + 1) mod (length (#2(#2 org))))` determines the next index in the schedule. First the current index is requested, this is incremented by 1, and afterwards this value is done modulo the length of the entire schedule. Furthermore, the time is updated indicating the last change.

```
fun updateOriginator(org: COriginator, time: CTime) =
    ((#1 org), ((((#1(#2 org)) + 1) mod (length (#2(#2 org
    )))), (#2 (#2 org)), time), (#3 org));
```

addableOriginator Similar to `removableOriginator`.

```
fun addableOriginator(xs) = removableOriginator(xs);
```

determineDurationUnrolled This is the unrolling of possible allocations of originators to activities. The function takes 3 arguments, the name of the activity ("C" for the first line), the name of the originator ("Role0" for the first line), and the list of work items. Based on this, the duration is determined using Yerkes-Dodson, i.e., for different intervals for waiting work items different durations are possible. The duration comprises of: (a) a time granularity factor

(0.0000027777777777777778 in the first line) to transform all the values to the same time granularity, (b) a translation factor of the distribution (0.0 in the first line), (c) a scaling factor (1.0 in the first line), and (d) the actual distribution ($normal(4.0, 1.0)$ in the first line).

```
fun determineDurationUnrolled("C", "Role0", xs) = if(50
   <= length(xs) andalso length(xs) < 70) then
    0.0000027777777777777778 * 0.0 + 1.0 *
    0.0000027777777777777778 * normal(4.0, 1.0) else if(10
     <= length(xs) andalso length(xs) < 20) then
    0.0000027777777777777778 * 0.0 + 1.0 *
    0.0000027777777777777778 * normal(4.0, 1.0) else if(20
     <= length(xs) andalso length(xs) < 50) then
    0.0000027777777777777778 * 0.0 + 1.0 *
    0.0000027777777777777778 * normal(2.0, 1.0) else if(0
   <= length(xs) andalso length(xs) < 10) then
    0.0000027777777777777778 * 0.0 + 1.0 *
    0.0000027777777777777778 * normal(5.0, 1.0) else 0.0|
determineDurationUnrolled("A", "Dennis", xs) = if(50 <=
    length(xs) andalso length(xs) < 70) then
    0.0000027777777777777778 * 0.0 + 1.0 *
    0.0000027777777777777778 * normal(4.0, 1.0) else if(10
     <= length(xs) andalso length(xs) < 20) then
    0.0000027777777777777778 * 0.0 + 1.0 *
    0.0000027777777777777778 * normal(4.0, 1.0) else if(20
     <= length(xs) andalso length(xs) < 50) then
    0.0000027777777777777778 * 0.0 + 1.0 *
    0.0000027777777777777778 * normal(2.0, 1.0) else if(0
   <= length(xs) andalso length(xs) < 10) then
    0.0000027777777777777778 * 0.0 + 1.0 *
    0.0000027777777777777778 * normal(5.0, 1.0) else 0.0|
determineDurationUnrolled("B", "Role0", xs) = if(50 <=
    length(xs) andalso length(xs) < 70) then
    0.0000027777777777777778 * 0.0 + 1.0 *
    0.0000027777777777777778 * normal(4.0, 1.0) else if(10
     <= length(xs) andalso length(xs) < 20) then
    0.0000027777777777777778 * 0.0 + 1.0 *
    0.0000027777777777777778 * normal(4.0, 1.0) else if(20
     <= length(xs) andalso length(xs) < 50) then
    0.0000027777777777777778 * 0.0 + 1.0 *
    0.0000027777777777777778 * normal(2.0, 1.0) else if(0
   <= length(xs) andalso length(xs) < 10) then
    0.0000027777777777777778 * 0.0 + 1.0 *
    0.0000027777777777777778 * normal(5.0, 1.0) else 0.0|
determineDurationUnrolled(_, _, _) = 0.0;
```

determineDuration Determines the duration for a work item, an originators, and a list of work items waiting.

```
fun determineDuration(w: CWorkItem, r:COriginator, xs:
    CWorkItems) = determineDurationUnrolled(getProperty("
    ActName", (#2 w)), getProperty("OrgName", (#3 r)), xs)
    ;
```

nextValue Is used to determine the next value for a variable. The function takes the current value, and the ValueMatrix encoding the Markov chain for the values of a variable.

```
fun nextValue(value, nil, xs: ValueMatrix) = nextValue(
    value, xs, xs) | nextValue(value, (oldValue, newValue,
     p)::valueMatrix: ValueMatrix, xs: ValueMatrix) = if(
    value = oldValue andalso bernoulli(p) = 1) then
    newValue else nextValue(value, valueMatrix, xs);
```

removeCaseId Removes the given case identifier from a list of case identifiers.

```
fun removeCaseId(a, b::xs) = if(a = b) then xs else b::
    removeCaseId(a, xs);
```

inList Returns whether an element is present in a list.

```
fun inList(a, nil) = false | inList(a, b::xs) = if(a = b)
     then true else inList(a, xs);
```

getLikeFactorUnrolled Similar to the `determineDurationUnrolled`, we take an activity, and an originator and if this originator may execute this activity, we return 1.0 else we return 0.0.

```
fun getLikeFactorUnrolled("C", "Role0") = 1.0|
getLikeFactorUnrolled("A", "Dennis") = 1.0|
getLikeFactorUnrolled("B", "Role0") = 1.0|
getLikeFactorUnrolled(_, _) = 0.0;
```

getLikeFactor Gets the relevant values from the attributes of an originator and a work item, and computes the like factor with them.

```
fun getLikeFactor(propW: Attributes, propR: Attributes) =
     getLikeFactorUnrolled(getProperty("ActName", propW),
    getProperty("OrgName", propR));
```

LikeFactor The `LikeFactor` gets the attributes from a work item and an originator, and computes the like factor on that.

```
fun LikeFactor((_, propertiesW): CWorkItem, (_, _,
    propertiesR): COriginator) = getLikeFactor(propertiesW
    , propertiesR);
```

maxLikeFactorUnrolled The `maxLikeFactorUnrolled` comes in two flavours based on push or pull allocation. The general idea is to compute the maximal like factor for between a list of work items and originators. If push allocation is desired, we take a work item and go through all the originators and find the most suitable one. After going through all the originators, if a suitable originator is found then we return this, else the next work item is checked. For pull allocation, we take an originator and go through the work items.

```
fun maxLikeFactorUnrolled(nil, _, _, d) = d |
    maxLikeFactorUnrolled(w::ws: CWorkItems, nil, R, d) =
    if(d <> 0.0) then d else maxLikeFactorUnrolled(ws, R,
    R, d) | maxLikeFactorUnrolled(w::ws: CWorkItems, r::rs
    : CLOriginator, R: CLOriginator, d: real) =
    maxLikeFactorUnrolled(w::ws, rs, R, Real.max(d,
    LikeFactor(w, r)));

fun maxLikeFactorUnrolled(_, nil, _, d) = d |
    maxLikeFactorUnrolled(nil, r::R: CLOriginator, W:
    CWorkItems, d) = if(d <> 0.0) then d else
    maxLikeFactorUnrolled(W, R, W, d) |
    maxLikeFactorUnrolled(w::ws: CWorkItems, r::rs:
    CLOriginator, W: CWorkItems, d: real) =
    maxLikeFactorUnrolled(ws, r::rs, W, Real.max(d,
    LikeFactor(w, r)));
```

maxLikeFactor The `maxLikeFactor` is a wrapper for the `maxLikeFactorUnrolled`, i.e., it duplicates the list of originators in case of push allocation, and duplicates the list of work items in case of pull allocation.

```
fun maxLikeFactor(_, nil) = 0.0 | maxLikeFactor(nil, _) =
    0.0 | maxLikeFactor(W: CWorkItems, R: CLOriginator) =
    maxLikeFactorUnrolled(W, R, R, 0.0);

fun maxLikeFactor(_, nil) = 0.0 | maxLikeFactor(nil, _) =
    0.0 | maxLikeFactor(W: CWorkItems, R: CLOriginator) =
    maxLikeFactorUnrolled(W, R, W, 0.0);
```

matchingPush This functions follows directly from the formalisation in Def. 54.

```
fun matchingPush(nil, nil, M: CMatching, R': CLOriginator)
    = M |
matchingPush(nil, r::R: CLOriginator, M: CMatching, R':
    CLOriginator) = M |
matchingPush(w::W: CWorkItems, nil, M: CMatching, R':
    CLOriginator) = matchingPush(W, R', M, R') |
matchingPush(w::W: CWorkItems, r::R: CLOriginator, M:
    CMatching, R': CLOriginator) = if(LikeFactor(w, r) =
    maxLikeFactor(w::W, r::R)) then (w, r)::M else
    matchingPush(w::W, R, M, R');
```

matchingPull This functions follows directly from the formalisation in Def. 55.

```
fun matchingPull(nil, nil, M: CMatching, W': CWorkItems) =
    M |
matchingPull(w::W: CWorkItems, nil, M: CMatching, W':
    CWorkItems) = M |
matchingPull(nil, r::R: CLOriginator, M: CMatching, W':
    CWorkItems) = matchingPull(W', R, M, W') |
matchingPull(w::W: CWorkItems, r::R: CLOriginator, M:
    CMatching, W': CWorkItems) = if(LikeFactor(w, r) =
    maxLikeFactor(w::W, r::R)) then (w, r)::M else
    matchingPull(W, r::R, M, W');
```

matching The matching function is an abstraction such that the core controller can be oblivious to the chosen allocation strategy. The first function is used in case of push allocation, the second in case of pull allocation.

```
fun matching(W: CWorkItems, R: CLOriginator, M: CMatching)
    = matchingPush(W, R, M, R);

fun matching(W: CWorkItems, R: CLOriginator, M: CMatching)
    = matchingPull(W, R, M, W);
```

updateWorkItemsMatching The `updateWorkItemMatching` removes all work items from the given list of work items which have been allocated to an originator.

```
fun updateWorkItemsMatching(W: CWorkItems, nil) = W |
    updateWorkItemsMatching(W: CWorkItems, (w, r)::M:
    CMatching) = updateWorkItemsMatching(rm w W, M);
```

updateOriginatorsMatching The `updateOriginatorsMatching` removes all originators for the given list of originators which have been allocated to a work item.

```
fun updateOriginatorsMatching(R: CLOriginator, nil) = R |
    updateOriginatorsMatching(R: CLOriginator, (w, r)::M:
    CMatching) = updateOriginatorsMatching(rm r R, M);
```

matchable Determines whether there is a match possible between any item of a list of work items and any item of a list of originators.

```
fun matchable(nil, _) = false | matchable(_, nil) = false
    | matchable(W:CWorkItems, R:CLOriginator) = if(
    maxLikeFactor(W, R) = 0.0) then false else true;
```

getDelay Gets the time an originator will be (un)available. This time is used to create a timed token solving the aforementioned problem with using time related information in a guard of a transition.

```
fun getDelay(r: COriginator) = List.nth(#2 (#2 r), (#1(
    #2 r) + 1) mod length(#2(#2 r)));
```

C Case study data

	A					B			
Event code	560532	560521	4634935	560519	560458	560429	1254625	560602	560613
540	✓	✓		✓	✓	✓	✓	✓	✓
550					✓		✓	✓	
550_1					✓				
560					✓		✓	✓	
590					✓		✓	✓	
600							✓	✓	
610							✓	✓	
630		✓	✓	✓	✓		✓	✓	✓
640							✓		
670					✓		✓		
680					✓		✓		
700							✓		
730					✓		✓	✓	✓
740								✓	
755		✓	✓	✓	✓		✓	✓	
760							✓		
765		✓	✓	✓	✓		✓	✓	
766								✓	
770		✓		✓	✓		✓	✓	✓
775		✓		✓			✓		
780_1								✓	
780_2								✓	
780_3								✓	
790								✓	

Table 7: The observed activities with originators who may execute them. Originators left of the double lines are for Municipality A, and right of the double lines are for Municipality B.

Replication	Sojourn	Processing	Queue	Wait
1	1612.22112327185	0.229856140212278	15.5622744792514	1612.22112327185
2	1585.61932783552	0.228572617003444	17.0522000129988	1585.61932783552
3	1605.23400773317	0.238762756736165	16.3098603649234	1605.23400773317
4	1592.74092825566	0.239019045259328	16.7547051120826	1592.74092825566
5	1617.99877967194	0.245221418321609	19.775160922796	1617.99877967194
6	1592.71788872882	0.233620505900653	13.9070242339544	1592.71788872882
7	1533.79078133865	0.26462493915919	9.70631572370623	1533.79078133865
8	1574.38748515519	0.233224084130629	9.82769773689952	1574.38748515519
9	1625.36617099035	0.239349941274681	10.8053074791598	1625.36617099035
10	1588.12035818254	0.234052894475994	12.0743869917517	1588.12035818254
11	1572.04701633281	0.226055154220116	10.8675773070279	1572.04701633281
12	1533.22985595812	0.225570078563927	9.57827092507011	1533.22985595812
13	1621.89331501793	0.235446098070214	11.2061671996374	1621.89331501793
14	1585.00546028374	0.230821112606567	10.3051076375256	1585.00546028374
15	1563.84038482556	0.227472317061724	10.2251477942419	1563.84038482556
16	1600.63977574163	0.247248346706576	15.3856883474569	1600.63977574163
17	1557.08760784248	0.228172871763337	16.8932711100201	1557.08760784248
18	1573.84184759888	0.228774540122265	16.0874152312756	1573.84184759888
19	1564.67593720558	0.232314366598844	14.2983842803352	1564.67593720558
20	1608.26127599673	0.235381849939744	14.8736839720352	1608.26127599673
21	1573.79275213965	0.231664709408134	17.4438137324801	1573.79275213965
22	1567.62619433745	0.232930874891936	14.6500160274213	1567.62619433745
23	1610.24446201573	0.234198882814128	15.6191325942308	1610.24446201573
24	1599.23911078275	0.238942483514538	13.9497726883193	1599.23911078275
25	1588.05974310647	0.238924431543654	12.4501679359481	1588.05974310647
26	1590.43837808957	0.230682141782928	14.4228594531347	1590.43837808957
27	1591.91406223369	0.228915482425434	14.1377340008558	1591.91406223369
28	1617.59600907846	0.237528681841463	14.7064967718827	1617.59600907846
29	1602.50260864363	0.237784855769712	13.0336081645067	1602.50260864363
30	1604.48359408511	0.286336085157448	12.0224713612777	1604.48359408511

Table 8: The results for Municipality A.

Replication	Sojourn	Processing	Queue	Wait
1	820.617834052972	0.194021693678179	19.0935821139414	820.617834052972
2	832.3172367993	0.189259575922633	12.6532286816937	832.3172367993
3	822.850684594141	0.18869537701677	12.3823711458945	822.850684594141
4	800.577148592002	0.180211635431849	12.7586195346657	800.577148592002
5	822.950611127349	0.187702957863273	11.795850918561	822.950611127349
6	805.217409118698	0.176279988876869	11.0402659988606	805.217409118698
7	845.420380944496	0.187702616444503	13.7044587633065	845.420380944496
8	825.587050928934	0.186510836974652	11.2337608153139	825.587050928934
9	837.610185751373	0.18770475895712	12.4774538021808	837.610185751373
10	812.333875562099	0.184610402547926	11.7332642473424	812.333875562099
11	852.207027577056	0.189139818578167	10.1081426002	852.207027577056
12	829.006471982173	0.180933528019977	11.3897446789364	829.006471982173
13	812.789094196666	0.180937552700843	12.9291763142793	812.789094196666
14	829.819984555536	0.195539667964852	13.0032896020216	829.819984555536
15	850.174771986015	0.188527730373374	12.1155205585953	850.174771986015
16	832.011651911417	0.178776856795859	9.72436354519957	832.011651911417
17	837.497366356126	0.182436606869071	11.2172656026651	837.497366356126
18	836.902257579932	0.183393691695624	12.226967604419	836.902257579932
19	824.02487169892	0.184531856042626	12.9201293531103	824.02487169892
20	834.927213713915	0.18557863749528	11.5673031217459	834.927213713915
21	809.939882657637	0.188307965542436	9.72635989240702	809.939882657637
22	813.071571090824	0.183926660511314	12.144391829827	813.071571090824
23	845.251353884779	0.187441881266197	12.3058741271281	845.251353884779
24	830.229991295176	0.183361853486166	12.7735889144577	830.229991295176
25	845.655719262369	0.191616646774431	10.7643620327028	845.655719262369
26	818.718792865131	0.182953046623038	10.3551927743589	818.718792865131
27	855.481703006381	0.200135668320262	13.9160293843327	855.481703006381
28	818.103804744154	0.186805174275945	11.3806484611856	818.103804744154
29	833.390833604063	0.185767839151276	11.9703297428391	833.390833604063
30	815.653899209819	0.184024166480548	14.5532453536797	815.653899209816

Table 9: The results for Municipality B.

Replication	1	2	3	4	5	6	7	8
1	1455.99	1582.76	1594.30	841.20	1486.01	1213.22	781.65	1229.93
2	1520.82	1612.45	1564.58	847.83	1539.03	1200.86	802.91	1215.85
3	1471.06	1574.19	1581.00	838.42	1475.57	1187.05	842.24	1232.01
4	1436.07	1577.33	1567.40	825.14	1466.96	1304.46	804.30	1188.84
5	1469.43	1506.14	1567.22	813.34	1555.50	1230.57	776.38	1200.36
6	1464.27	1529.17	1562.33	815.51	1512.90	1196.32	790.98	1235.66
7	1524.01	1562.49	1598.25	791.43	1495.05	1169.80	810.43	1204.22
8	1460.09	1587.19	1562.63	802.96	1491.73	1238.81	811.51	1339.30
9	1560.93	1573.69	1565.48	821.01	1494.98	1241.78	789.52	1229.11
10	1489.40	1526.58	1585.59	837.46	1494.10	1212.70	805.53	1248.36
11	1502.43	1579.15	1576.78	793.69	1484.00	1237.18	819.58	1238.89
12	1523.95	1573.34	1564.88	838.27	1496.94	1240.73	828.11	1205.80
13	1508.51	1610.34	1592.09	779.28	1509.99	1221.86	797.09	1185.47
14	1488.06	1583.75	1583.12	842.90	1479.48	1259.14	795.57	1211.77
15	1499.11	1627.86	1564.68	806.40	1462.33	1279.69	816.09	1301.11
16	1522.56	1570.09	1557.85	789.72	1535.84	1164.07	782.85	1194.94
17	1505.44	1542.66	1544.42	760.03	1464.14	1192.59	837.70	1210.59
18	1445.92	1545.28	1556.61	783.28	1452.05	1190.30	804.58	1187.28
19	1499.13	1560.00	1576.10	783.25	1474.02	1198.89	812.39	1208.50
20	1483.05	1579.07	1579.58	798.95	1474.32	1163.52	800.07	1251.44
21	1527.86	1623.60	1571.14	800.17	1494.14	1174.17	815.61	1236.34
22	1466.18	1571.82	1571.11	797.35	1517.23	1230.87	801.29	1201.38
23	1449.32	1589.89	1528.38	811.28	1511.64	1207.62	782.63	1233.12
24	1487.21	1588.08	1627.24	799.24	1507.07	1200.39	784.56	1213.29
25	1447.66	1617.67	1560.65	780.83	1523.26	1228.60	769.93	1229.88
26	1495.07	1532.75	1579.40	772.18	1481.47	1211.29	791.97	1173.09
27	1460.44	1505.35	1561.31	817.10	1485.74	1223.98	770.48	1232.99
28	1503.63	1636.83	1547.46	786.82	1535.84	1290.68	846.08	1240.51
29	1493.19	1551.77	1552.75	807.35	1495.48	1202.44	807.08	1229.57
30	1502.63	1553.21	1582.90	815.64	1486.37	1206.18	831.05	1225.39
540	A	A	A	B	A	B	B	B
630	A	A	B	A	B	A	B	B
770	B	A	A	B	B	A	B	A
Avg	1488.78	1572.48	1570.91	806.60	1496.11	1217.32	803.67	1224.50
Std. dev.	29.55	33.82	18.58	22.93	24.81	34.80	20.24	33.10

Table 10: The results for the case study (sojourn times in hours).

References

1. Schunselaar, D.M.M., Verbeek, H.M.W., Aalst, W.M.P. van der, Reijers, H.A.: Creating Sound and Reversible Configurable Process Models Using CoSeNets. In Abramowicz, W., Kriksciuniene, D., Sakalauskas, V., eds.: BIS. Volume 117 of Lecture Notes in Business Information Processing., Springer (2012) 24–35
2. Rosemann, M., Aalst, W.M.P. van der: A Configurable Reference Modelling Language. Information Systems **32**(1) (2007) 1–23
3. La Rosa, M., Dumas, M., Hofstede, A.H.M. ter, Mendling, J.: Configurable multiperspective business process models. Inf. Syst. **36**(2) (2011) 313–340
4. Gottschalk, F.: Configurable Process Models. PhD thesis, Eindhoven University of Technology, The Netherlands (2009)
5. Gottschalk, F., Aalst, W. M. P. van der, Jansen-Vullers, M.H., Verbeek, H.M.W.: Protos2CPN: Using Colored Petri Nets for Configuring and Testing Business Processes. International Journal on Software Tools for Technology Transfer **10**(1) (2008) 95–110
6. Aalst, W.M.P. van der, Nakatumba, J., Rozinat, A., Russell, N.: Business Process Simulation. In Brocke, J., Rosemann, M., eds.: Handbook on Business Process Management 1. International Handbooks on Information Systems. Springer Berlin Heidelberg (2010) 313–338
7. La Rosa, M., Lux, J., Seidel, S., Dumas, M., Hofstede, A.H.M. ter: Questionnaire-driven Configuration of Reference Process Models. Advanced Information Systems Engineering **4495** (2007) 424–438
8. Hallerbach, A., Bauer, T., Reichert, M.: Capturing Variability in Business Process Models: The Provop Approach. Journal of Software Maintenance and Evolution: Research and Practice **22**(6-7) (November 2010) 519–546
9. Gagne, D., Shapiro, R.: BPSim 1.0. http://bpsim.org/specifications/1.0/WFMC-BPSWG-2012-01.pdf (Feb 2013)
10. Verbeek, H.M.W., Aalst, W.M.P. van der: Woflan 2.0: A Petri-Net-Based Workflow Diagnosis Tool. In: ICATPN. (2000) 475–484
11. Gross, D., Shortle, J.F., Thompson, J.M., Harris, C.M.: Fundamentals of Queueing Theory. 4th edn. Wiley-Interscience, New York, NY, USA (2008)
12. Laguna, M., Marklund, J.: Business Process Modeling, Simulation, and Design. Prentice Hall, New York (2004)
13. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer (2009)
14. Chiola, G., Franceschinis, G., Gaeta, R., Ribaudo, M.: GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic Petri nets. Performance Evaluation **24**(1 - 2) (1995) 47 – 68
15. Bertoli, M., Casale, G., Serazzi, G.: JMT: performance engineering tools for system modeling. SIGMETRICS Perform. Eval. Rev. **36**(4) (2009) 10–15
16. Dingle, N.J., Knottenbelt, W.J., Suto, T.: Pipe2: A tool for the performance evaluation of generalised stochastic petri nets. SIGMETRICS Perform. Eval. Rev. **36**(4) (March 2009) 34–39
17. Zimmermann, A., Freiheit, J., German, R., Hommel, G.: Petri net modelling and performability evaluation with timenet 3.0. In Haverkort, B.R., Bohnenkamp, H.C., Smith, C.U., eds.: Computer Performance Evaluation.Modelling Techniques and Tools. Volume 1786 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2000) 188–202

18. Berthomieu, B., Vernadat, F.: Time Petri Nets Analysis with TINA. In: Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on. (2006) 123–124
19. Westergaard, M.: Access/CPN 2.0: A High-Level Interface to Coloured Petri Net Models. In Kristensen, L.M., Petrucci, L., eds.: Petri Nets. Volume 6709 of Lecture Notes in Computer Science., Springer (2011) 328–337
20. Brand, N., Kolk, H. van der: Workflow Analysis and Design. Kluwer Bedrijfswetenschappen, Deventer (In Dutch) (1995)
21. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer (2013)
22. Netjes, M., Mansar, S.L., Reijers, H.A., Aalst, W.M.P. van der: Performing Business Process Redesign with Best Practices: An Evolutionary Approach. In Filipe, J., Cordeiro, J., Cardoso, J., eds.: ICEIS (Selected Papers). Volume 12 of Lecture Notes in Business Information Processing., Springer (2007) 199–211
23. Netjes, M., Reijers, H.A., Aalst, W.M.P. van der: The PrICE Tool Kit: Tool Support for Process Improvement. (2010)
24. Hoesch-Klohe, K., Ghose, A.: Business Process Improvement in Abnoba. In Maximilien, E.M., Rossi, G., Yuan, S.T., Ludwig, H., Fantinato, M., eds.: ICSOC Workshops. Volume 6568 of Lecture Notes in Computer Science. (2010) 193–202
25. Essam, M.M., Mansar, S.L.: Towards a Software Framework for Automatic Business Process Redesign. ACEEE International Journal on Communication **2**(1) (March 2011) 6
26. Măruşter, L., Beest, N.R.T.P.: Redesigning business processes: a methodology based on simulation and process mining techniques. Knowledge and Information Systems **21**(3) (2009) 267–297
27. Netjes, M., Reijers, H.A., Aalst, W.M.P. van der: On the Formal Generation of Process Redesigns. In Ardagna, D., Mecella, M., Yang, J., eds.: Business Process Management Workshops. Volume 17 of Lecture Notes in Business Information Processing., Springer (2008) 224–235
28. Kalenkova, A.A.: Application of if-conversion to verification and optimization of workflows. Programming and Computer Software **36**(5) (2010) 276–288
29. Kalenkova, A.A.: An algorithm of automatic workflow optimization. Programming and Computer Software **38**(1) (2012) 43–56
30. Kaiya, H., Morita, S., Kaijiri, K., Hayashi, S., Saeki, M.: Facilitating business improvement by information systems using model transformation and metrics. In Kirikova, M., Stirna, J., eds.: CAiSE Forum. Volume 855 of CEUR Workshop Proceedings., CEUR-WS.org (2012) 106–113
31. Wynn, M.T., Dumas, M., Fidge, C.J., Hofstede, A.H.M. ter, Aalst, W.M.P. van der: Business Process Simulation for Operational Decision Support. In Hofstede, A.H.M. ter, Benatallah, B., Paik, H.Y., eds.: Business Process Management Workshops. Volume 4928 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2008) 66–77
32. Hofstede, A.H.M. ter, Aalst, W.M.P. van der, Adams, M., Russell, N., eds.: Modern Business Process Automation: YAWL and its Support Environment. Springer (2010)
33. Vanhatalo, J., Völzer, H., Koehler, J.: The Refined Process Structure Tree. In Dumas, M., Reichert, M., Shan, M.C., eds.: BPM. Volume 5240 of Lecture Notes in Computer Science., Springer (2008) 100–115
34. Aalst, W. M. P. van der, Hee, K. M. van: Workflow Management: Models, Methods, and Systems. The MIT Press (January 2002)

35. Yerkes, R.M., Dodson, J.D.: The Relation of Strength of Stimulus to Rapidity of Habit-Formation. Journal of Comparative Neurology and Psychology **18**(5) (1908) 459–482
36. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. Prentice Hall PTR (September 1999)
37. Aalst, W.M.P. van der, Adriansyah, A., Dongen, B.F. van : Replaying History on Process Models for Conformance Checking and Performance Analysis. Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery **2**(2) (2012) 182–192
38. Fishman, G.S.: Grouping Observations in Digital Simulation. Management Science **24**(5) (1978) pp. 510–521
39. Kung, H.T., Luccio, F., Preparata, F.P.: On finding the maxima of a set of vectors. J. ACM **22**(4) (1975) 469–476
40. Aalst, W.M.P. van der, Rosemann, M., Dumas, M.: Deadline-based escalation in process-aware information systems. Decision Support Systems **43**(2) (2007) 492 – 511