M.Sc. Thesis
Master of Science and Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Generation of Business Process Models for Benchmarking Purposes

A metrics-based approach

Anders Thestrup Rikvold (s121919)

Kongens Lyngby 2017

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

Many systems perform tasks that require process models as input. For testing the performance of these systems, it is useful to provide a benchmark consisting of a collection of process models. However, constructing a collection with certain desired characteristics from real-life models can be difficult and time consuming. By automatically generating the collection, it can be constructed faster and with more ease, but this puts high demands on the generation algorithm providing control of the characteristics of the generated collection. Existing algorithms either provide little direct control over these characteristics, or require a smaller collection of models with the same characteristics to be available. This work proposes a process model generator that takes the desired characteristics as input in the form of process model metrics, providing direct control of the characteristics of the generated collection. The main idea of the algorithm is to generate each model by incrementally reducing the difference between its current and desired characteristics, until a satisfactory solution is met. This approach produces collections with characteristics that largely resemble the desired ones. While it does have certain shortcomings, such as providing comprehensive support for only a small set of metrics, which makes it impractical for use in real-life situations, it proposes possible solutions for each of these, and is a significant first step towards designing a practical algorithm.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a Master of Science in Engineering degree in Computer Science and Engineering.

Kongens Lyngby, December 30, 2017

Anders Thestrup Rikvold (s121919)

Acknowledgements

I would like to express my deep appreciation for my supervisors, Andrea Burattin and Barbara Weber, for their great guidance and constructive feedback during the project. A special thanks should be given to Andrea Burattin for taking the time and effort of providing valuable advice and guidance in the day-to-day work on the thesis, and for his encouragements and technical expertise in the research field, which have been of great significance to the end result. I would also like to thank the Technical University of Denmark for providing the framework for conducting the thesis work.

Finally, I wish to thank my family for supporting me throughout my studies and the thesis work, and Per Arne Rikvold for so generously offering to proofread the thesis during its final stages.

Contents

CHAPTER 1

Introduction

Business process models are an important tool for many organizations, and aid in tasks such as documentation of procedures, enterprise integration, performance analysis and improvement [2], and act as reference models for the creation of similar processes [12]. Many systems that operate on process models also exist, such as model editors, simulation tools for generating logs, or repositories that organize a large set of models [30, 13]. These systems often have advanced algorithms for performing operations involving process models. Model editors, for instance, might analyze a model to check for errors [15], and alert the designer if it discovers any. Model repositories might have a method for doing similarity searches with a specific model, to find other similar models [24, 28, 42, 14].

When developing such algorithms, it is important to test them, to see how well they perform. This could mean testing their correctness on a certain set of process models, but could also mean to test their performance, specifically their average performance, by subjecting them to a benchmark made up of a collection of process models exhibiting relevant characteristics [21]. For an error checking algorithm, for example, it might be desirable to test it on a range of different process models of a large size, to determine the average performance as the size of the models in question grows. If too small models are chosen, the benchmark will not be relevant to what it is supposed to be testing, and if too similar models are chosen, it will be difficult to draw any conclusion about how the algorithm might perform on different models of the same size, so it might not be fair to other systems using the same benchmark. It is therefore important to choose the right process model collection when creating a quality benchmark.

In addition to systems that work on directly on them, process models can also be useful within process mining [1], which is a discipline that analyzes logs from business processes. This can involve, for example, determining how well a process conforms to its definition in practice, i.e. process conformance checking, with the ultimate goal of improving the model describing the process. When working with process mining algorithms, it is often useful to have the prescribed process that a set of logs belong to, so it can be used as a reference. Such a reference is also often used as a ground truth which the results of the algorithm can be compared to. When evaluating a process discovery algorithm, which is an algorithm that attempts to reconstruct a process definition from its logs, having a reference to compare the discovered model to is very important. Referring to a ground truth is often vital to the evaluation of

the quality of a process mining algorithm [1].

The process used as a ground truth generally can not just be any process model. Like with algorithms that operate on process models, process mining algorithms are tested or benchmarked on logs that come from processes that exhibit certain desired characteristics. It might be of particular interest to test a process discovery algorithm on logs generated from a process that contains deadlocks, for example. This also means that choosing process models that exhibit certain characteristics is as important in process mining, as it is in other algorithms.

There are mainly two ways of obtaining process models to use for testing or benchmarking an algorithm, both with their advantages and disadvantages [32]. One is to select the models from an existing source, such as a publicly available process model repository. This allows developers to quickly attain usable models that usually represent real-life processes, but comes with the drawback of reduced flexibility and control over what kind of models are used. For example, the developer might have found models that display the desired characteristics, but the set is not of adequate size, or is otherwise inadequate for drawing conclusions about the algorithm's performance. This means that the developer will have to look for other models to make up for current shortcomings, which can be very time-consuming, if even possible, and might lead to choosing sub-optimal models. This problem is exaggerated by the fact that many models are closely guarded trade secrets of their respective companies, or contain personal or sensitive information, so the number of publicly available real-life process models is limited [43].

Another way of obtaining suitable process models is to automatically generate them. This approach is however strongly dependent on the generation algorithm used, and how well it can generate process models that exhibit the desired characteristics. There are several proposed solutions for such process model generators, such as the basic one included in the Process Log Generator (PLG) [5]. This generator defines a set of basic patterns that are iteratively chosen during generation of a model, based on a predetermined probability assigned to each. While simple, the generator does not provide the user with a high degree of control of the characteristics that the generated process models will display, as the generation process only takes the assigned probabilities as input. The work of van Hee et al. [40] remedies this by determining the input probabilities by decomposing a collection of models that display the desired characteristics. While they achieve higher control of the generated models by faithfully reproducing the characteristics of the reference collection, the method requires users to be in possession of a reference collection with the desired characteristics, which is not always the case, or searching for one might be extensively time-consuming or difficult. This is also true if the characteristics of the collection have to be adjusted slightly from an available collection.

This work aims to create a generation method for business process model collections that provides increased control of the characteristics of generated collections, by taking the desired characteristics of the generated collections as input, in the form of metrics. From these metrics, the probabilities for choosing each basic pattern will be determined, and continuously updated for each iteration of the generation, based

on the current state of each generated model compared to the desired one. In addition to eliminating the need for having an available model collection with the desired characteristics, this method puts the user in direct control of what characteristics generated collections will display.

The rest of the thesis presents this approach in the following manner. Chapter 2 puts the proposed algorithm in the context of existing research in the field, and chapter 3 presents a number of preliminary concepts that are referenced in the rest of the thesis. Chapter 4 presents the algorithm and central concepts, before chapter 5 provides an overview of its implementation. From this implementation, the results presented in chapter 6 are obtained, which also provides a discussion of several aspects of its performance. Finally, chapter 7 provides a summary of the thesis and lists suggestions for future research into the topic.

This page has been intentionally left blank

CHAPTER 2

Related Work

There have been several previous works tackling the problem of generating business process models. Some of these generators are quite basic, and are only parts of a more comprehensive research focus. One such generator is the one included in the Process Log Generator (PLG) [6], currently in its second iteration as PLG2 [5], which proposes a generator for models that are used to generate logs from, serving as a ground truth for the logs. This generator works by defining a context-free grammar producing process models, and deriving a set of basic patterns from the context-free grammar. During execution, these patterns are recursively produced in the model until no more symbols are left. What patterns are chosen depends on a set of predetermined weights of each pattern, which dictate the probabilities of choosing each one. A similar generation algorithm is the one in [24], which randomly chooses one of several refinement rules as defined in [10], and uses the generated models in an indexing algorithm. While being able to quickly generate large and random process model sets, they provide little direct control over what characteristics the generated model collections will have.

A similar process model generator is defined in [39]. However, this generator is specifically designed to generate process models for creating benchmarks, and thus provides some more control of the generated characteristics by allowing users to choose between different *classes* of workflow nets. These can be either *Jackson-nets*, *state machine* nets, *marked-graph* nets and *free-choice* nets, and decide what patterns, or *construction rules*, will be available. The framework of the generator also allows for a subset of workflow nets that have certain characteristics to be chosen from the superset of all generated models. Despite providing increased control of the output characteristics, the algorithm still requires the user to specify the probabilities of applying each construction rule, which in the end determines what characteristics the workflow net will have.

The work in [40] extends the work of [39] by analyzing existing process model collections to find their *generation parameters*, i.e. the probabilities assigned to each construction rule in order to generate similar models. By analyzing a collection of nets with known characteristics, this allows the generation of a bigger collection of nets with similar characteristics. This is a major improvement compared to previous work, but still only provides indirect control of the generated characteristics, since it requires a collection of nets with the desired characteristics on-hand. In contrast, the method proposed in this thesis removes this middle link, providing direct control of

the produced characteristics, by allowing users to directly specify them.

The work in [43] and [44] also deals with process model generation, but their focus is on the labels and naming of the components in the models, rather than their structure, which is the focus of this thesis.

CHAPTER 3

Background

The thesis is targeted towards readers that have a background within computer science, and the rest of the work will use this as a baseline. As such, it assumes that readers have an understanding of basic concepts within fields such as graph theory, algorithms, process and software modeling techniques, and principles of software design.

The thesis does however require certain preliminaries specific to the field of business process modeling, which can not be expected by everyone with a computer science background. In order to appeal to such readers, this section presents some basic concepts that the rest of the thesis assumes the reader to be familiar with. The concepts are not presented in-depth, but is meant to give the reader a basic understanding of the concepts - just enough to be able to fully understand references to the material later in the thesis.

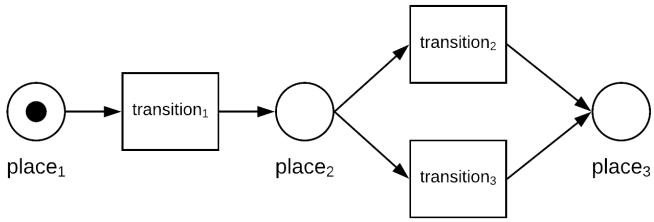
3.1 Process Models

Workflow patterns, and in turn workflow models, form the basis of business process models. These basic workflow patterns describe basic control-flow, which can be used to model real-life processes. Unless otherwise stated, the information presented in this section is reproduced from [27].

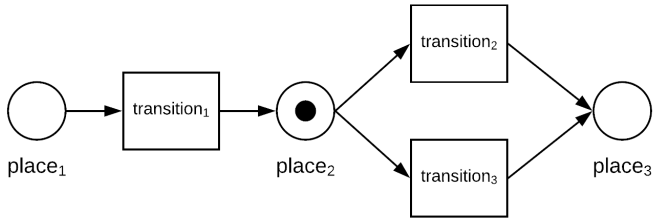
In order to present such a basic workflow pattern, a language is needed. Many works use some form of a Petri net, which is very suitable for this task due to their simple notation and control-flow focus. However, there are often several other aspects of real-life processes that modelers want to represent. These can include information about what certain activities or tasks of a process do, who performs them, or what starts them. This means that modelers often want a language with more expressiveness than Petri nets to describe their processes, of which several exist. This thesis will mostly focus on Business Process Modeling Notation (BPMN), but other languages, such as Event-driven Process Chain (EPC) and Business Process Execution Language (BPEL), also exist.

3.1.1 Petri Nets

A Petri net is one of the most prominent techniques for capturing the behavior of concurrent systems, and has been adopted by several works to describe workflows, such as [4, 36, 40]. A Petri net is defined as a graph, which consists of places, transitions and directed arcs between these. Since arcs only ever connect from a place to a transition, or the other way around, it is a bipartite graph. An example of a Petri net can be seen in figure 3.1.



(a) A Petri net with a token at $place_1$



(b) The Petri net of 3.1(a) after firing $transition_1$, transferring the token to $place_2$

Figure 3.1: An example of Petri nets, where the vertices have been labeled with their types, and are connected by arcs.

In order to model the flow of a process net, it is possible to execute it. This is done by defining the concept of *tokens*, which can be placed any any place in the net, though preferably some form of start place, initially. The net can then be executed by *firing* a transition, which moves the token along the net. A more thorough description of Petri nets, along with their formal definition, can be found in [27].

3.1.2 BPMN

The Business Process Model Notation, or BPMN for short, is the industry standard for modeling business processes. While the language contains many different elements,

the following basic control-flow objects can be used to capture the central aspects of many models.

Activities The most basic component of this kind is a *task*. Other components of this category are sub-processes and call activities, but these are not in focus in this work. Therefore, the terms *task* and *activity* will be used interchangeably to mean a task.

Gateways These elements signal the splitting or joining of execution paths. The elements that will be used in this work are *parallel* and *exclusive* gateways, also referred to as AND- or XOR-gateways. Other gateways include event-based gateways, that directs process execution based on an event happening, or inclusive gateways, where conditions on choosing all paths are evaluated.

Events These elements indicate something happening. This work will focus on *start*- and *end*-events, but an intermediate event signalling an event occurring during process execution is also defined.

Sequence flow This element, indicated as an arrow with a solid line between flow components in the model, indicate the possibility of transitioning from one component to another.

Aside from the above listed objects, BPMN also has *swim lanes* and *artifacts*, which give it the ability to capture additional aspects of real-life business processes. Swim lanes allow for modeling the responsibilities of a process, i.e. who is responsible for performing the contained tasks, while artifacts, such as data objects and annotations, can be used to add extra information to the model. It also lists other types of connections beside sequence flows, such as message flows. However, the focus of this work will mainly be with regards to the above listed control-flow objects. With only these basic objects, it is easy to draw parallels between the resulting BPMN models and basic workflow models expressed as Petri nets.

An example of a BPMN model using basic control-flow objects can be seen in figure 3.2. As shown in [27], a BPMN model using these elements usually has a basic Petri net workflow equivalent.

3.1.3 Behavioral Properties

It is often useful to describe workflow models in terms of their behavioral properties. These properties are not to be confused with process model characteristics, as defined in section 3.2. The most relevant properties for this work will be presented in this section. In general, behavioral properties of models can be divided into the following four properties.

Reachability This kind of property is about a system's possibility of reaching a certain state. In the context of business process models, it makes sense to

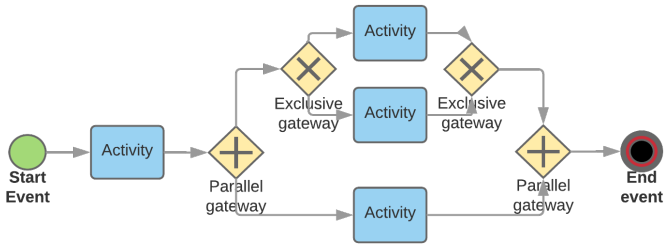


Figure 3.2: Example of a BPMN diagram. Each of the flow objects used have been labeled with their respective types. This work will focus on the components used in this figure.

talk about the reachability of an activity, which means that if the activity is contained in the model, it should be possible to be reached during execution.

Safety This kind of property is about the absence of something undesirable happening. In the context of business process models, the *absence of deadlocks* is an important safety property. A deadlock is a situation where the system gets stuck, such that it is unable to continue. In terms of a BPMN model, this could occur if a parallel gateway join is waiting for two preceding activities to be reached, but an exclusive gateway split earlier in the flow means one of the activities will never be reached.

Liveness By ensuring liveness properties of a system, it is ensured that the system progresses. A common such property is the guaranteed termination of a system, e.g. the system reaching an end-event in a BPMN model.

Fairness The fairness property implies that a choice is not ignored forever. This means for example that if an xor gateway is part of an infinite loop, all the choices will eventually be taken, and none ignored forever.

In addition to these properties, [38] discusses a *soundness property*, stating that a sound process ensures (1) an option to complete, i.e. reaching an end-state, (2) completing properly, i.e. there are no more states executing in the model when termination happens, and (3) there are no dead transitions, i.e. any arbitrary activity can be reached. They refer to this kind of soundness as *classical soundness*. Ensuring property (1) implies fairness, assuming that all possible choices in a mutual exclusion gateway split will always eventually be chosen. It also implies safety by the absence of deadlocks, and that the system will always eventually terminate. The second aspect of soundness implies that no other parts of the system is running when the model terminates, and the third one implies reachability for every activity in the process model.

3.1.4 Constructing Process Models

The existing process model generators mentioned in chapter 2 use a variety of techniques for actually generating their process models. While there exists some work within the related field of graph synthesis (such as [3]), the concepts have only occasionally been applied to workflow net generation. One such application is in [41], where the process generation techniques of [29] were extended to business process models.

Another generation technique relies on the construction rules studied by Berthelot in [4] and Suzuki et al. in [36]. In [19], nets constructed from these rules, with the exception of a loop addition rule, are called *Jackson-nets*. Such nets ensure the *soundness* property, which is proven in [18]. The concept of *soundness* is explained in section 3.1.3. In [39], an extension of this construction rule set is used to generate workflow nets, with which they also prove that it is possible to construct any Petri net, but which does not preserve soundness.

The basic rules studied in [4, 36] are called *Murata rules*, and are defined for Petri nets. The rules are listed as R1-R6 in table 3.1, and mainly add places or transitions to the model, duplicating them, or adding loops. In order to construct a model from such construction rules, they have to be applied several times to a model, where the model starts as a singular net of just one node. By consecutively applying the rules at different places of the net, it will gradually grow bigger and more complex, until the desired has been constructed.

In [39, 40], an extra set of rules, *bridge rules*, are added to the Murata rules. As the name suggests, these rules ”bridge” the model by adding a connection between two separate parts of the model. While these new rules do not necessarily preserve the soundness of the model, they allow any Petri net to be constructed.

Other generation techniques include generating workflow models from a stochastic context-free grammar, as is done by PLG [5]. This approach is explained in further detail in section 3.3.

3.2 Metrics as Process Model Characteristics

In general, a metric is a standard of measurement, used to quantify a property of something. Unlike a measure, a metric is usually not something that is measured directly, but is obtained by combining one or more measurements. For example, a metric for the customer satisfaction of a business could be the average rating, which is a combination of the total value of the ratings divided by the number of ratings, given to the business by customers through a web portal such as the one discussed in [23]. These metrics are found throughout the sciences, and play an important role in quantifying properties of research items. For the remainder of this thesis, there will not be made any distinction between a metric and a measure when discussing process models, which will be referred to as just *metrics*. Also, the terms *characteristic* and *metric* will be used interchangeably to refer to the characteristics of a process

Table 3.1: The construction rules used in [39, 40]. They have been freely reproduced from their formal definitions in [40].

Rule	Description
R1: Place refinement	Turns a place into two places connected by a transition
R2: Transition refinement	Turns a transition into two transitions connected by a place
R3: Loop addition	Adds a new transition and connects it to a place of the model through a two-way arc
R4: Place duplication	Duplicates a place along with its arcs to transitions
R5: Transition duplication	Duplicates a transition along with its arcs to places
R6: Arc refinement	Adds a sequence of a transition and a place to another sequence of a transition and a place
R7: Place bridge	Adds a place connected to two transitions, at any two positions of the net
R8: Transition bridge	Adds a transition connected to two places, at any two positions of the net
R9: Arc bridge	Adds an arc between a transition and a place, at any two positions of the net

model. This implies the assumption that metrics can fully describe the characteristics of a model, which may or may not hold true, depending of what definition of a characteristic is used. However, for the purposes of this thesis, this assumption will be made.

Metrics have been extensively used in several areas of research related to business process modelling. One of these areas is network analysis, meaning the field of structural theory within applied graph theory [33]. As this work focuses on BPMN-models, which can be seen as a special kind of graph, a lot of work from this area can be applied to business process models. Metrics used within this field can be, for example, the density or distance of a network. Density is defined as the number of edges in a graph, compared to the number of possible edges in the same graph. In the case of a process model, the density could be an indicator of the complexity of the process flow, where the more connections a process model has, the more possible flows; hence a higher flow complexity. Likewise, the distance in a network, i.e. the number of edges in a shortest path between two nodes, can be used to tell the length of a process model, by looking at the distance from the start to end node.

Other metrics used in network theory can also be applied to business process models, such as degree, centrality or connectivity. A further exploration of metrics for business process modelling, as well as for this project in particular, will be provided in section 3.2.1.

When using metrics in the field of business process modeling, which metrics to focus on is highly dependent on the purpose of the research. If the purpose is related to improving process model design, it makes sense to look at metrics related to the quality of process models, or *quality metrics*. What makes up such metrics, i.e. what metrics correlate with the subjective construct of quality, is a matter of research in itself, where one might have to resort to heuristics or logical arguments, rather than empirical knowledge. Examples of metrics that have been linked to quality are the number of distinct paths in the model, hierarchy levels showing the amount of nesting, or parallelism [35]. Several works link the quality and likelihood of errors in a model to its complexity, with the assumption that higher complexities make models harder to comprehend for designers, and thus lead to a lower quality and higher likelihood of errors [17, 33]. This also means that metrics related to model complexity, *complexity metrics*, are a subject of study in several works [9, 8, 31]. The work of Mendling [33] tries to make a distinct connection between the complexity of a model and its likelihood of errors, and concludes that there is indeed a significant correlation.

3.2.1 Specific Metrics

In [33], Mendling divides process complexity metrics into six categories, for presentation purposes. This work will use the same categories, which are as follows.

Size Includes metrics that affect the size of the process model. An obvious example is the number of nodes in the process model, which for most BPMN models means the number of activities and gateways. However, it also includes metrics such as its *diameter*.

Density While density for a graph is usually defined as the ratio of the actual number of edges to the total number of edges, it will for this work be used to refer to any metric that describes the number of nodes related to the number of arcs, or connectors. One such metric is, of course, its *density*. It however also includes the *coefficient of network connectivity*, i.e. the ratio of arcs to nodes, the *average connector degree*, i.e. the average number of nodes a connector connects to, and the *maximum degree*, i.e. the highest number of nodes a connector connects to.

Partitionability This term is used to refer to the degree to which a model can be partitioned into sub-components, which can be seen in isolation from other parts of the model. This could be the number of SISO-parts in the model, or how many nesting ”blocks” of split/join gateways it has. In this category, Mendling included metrics such as *separability*, which is the ratio of cut-vertices (articulation points) to all vertices of a model, *sequentiality*, which is the ratio of edges between two sequential nodes to all edges in the model, and the *depth*, which is the maximum depth of the nesting in the graph.

Gateway Interplay While this category was originally called *connector interplay* by Mendling, it has been renamed for the purpose of this work, as a *connector* is mainly an EPC-related term. As the name suggests, it contains metrics related to gateways, and their relation to each other. One metric in this category is the *connector mismatch*, which measures the number of split-gateways that do not have a corresponding join-gateway. Another one is the *Control Flow Complexity*, or CFC, which considers the all possible states of the model a designer has to take into account, which is dependent on the number and kinds of splits in the model.

Cyclicity Indicates the degree to which the model is cyclic, i.e. contains cyclic constructs such as a loop. Melding proposes a metric *cyclicity* which is the ratio of nodes in a cycle to the total number of nodes in the model. Another metric was proposed in [35], which counts the number of cycles instead.

Concurrency This category includes metrics that try to capture the number of paths that need to be synchronized. Mendling suggests doing this by summing the out-degree of all OR- and AND-splits in the model, while others have suggested counting the maximum number of parallel paths that can be executing at the same time, possibly related to the total number of nodes.

More specifically, [33] lists the following metrics. While some metrics might differ from the ones as defined in the work by Mendling, the following definition of the metrics will be used in this thesis. One important reason for the differences is the CFG used by PLG, of which the same framework is used by the work in this thesis, which does not include OR-gates, so the metrics have been adapted to reflect this, in the cases where this makes a difference.

The following definitions are used in the tables 3.2 and 3.3.

- G: Process model graph
- N: Nodes in G
- T: Activities in G
- A: Arcs or connections in G
- C: Gateways (connectors) in G
- L: the set of distinct cycles in G. For each $l \in L$, l: $\{n \in N\}$, such that each element in L is a set containing certain nodes in G.

3.3 PLG - Process Log Generator

In [6], Burattin et al. propose a framework for generating business process models and their execution logs. The framework is expanded in [5], which introduces its

Table 3.2: The first half of the metrics defined in [33], which are the focus of this work. See also table 3.3.

Category	Metric	Definition
Size	Number of nodes	$S_N(G) = \|N\|$ The total number of nodes in the graph
	Number of activities	$S_{act} = \|T\|$ The number of activities in G. Not defined in [33].
	Number of gateways	$S_C = \|C\|$ The number of gateways in G. Not defined in [33].
	Number of AND-gateways	$S_{and} = \|C_{and}\|$ The number of AND-gateways in G. Not defined in [33].
	Number of XOR-gateways	$S_{xor} = \|C_{xor}\|$ The number of XOR-gateways in G. Not defined in [33].
	Diameter	$diam(G)$ Length of the longest part from the start to end node of G
Density	Density	$\Delta(G) = \frac{\|A\|}{\|N\|*(\|N\|-1)}$ The ratio of the number of arcs to the maximum possible number of arcs in the model
	Coefficient of connectivity	$CNC(G) = \frac{\|A\|}{\|N\|}$ The ratio of arcs to nodes in the model
	Average degree of connectors	$d_C(G) = \frac{1}{\|C\|} * \sum_{c \in C} d(c)$ The average out-degree of a gateway
	Maximum degree of connectors	$\hat{d_C}(G) = max\{d(c) \| c \in C\}$ The maximum out-degree of any gateway

second iteration, PLG2. The information on the framework presented in this section is largely a reproduction of information presented in these two works. The term PLG will be used to refer to both versions of the framework.

The generation algorithm proposed in this thesis is an extension of the PLG process generation framework, and therefore has many things in common with it. For this reason, the framework of PLG will be referred to frequently, and a more in-depth explanation of the framework will be provided in this section.

The generation engine proposed by PLG uses a proposed context-free grammar (CFG) as the basis for generating process models, where all producible process models are the result of the production rules P of PLG, and can be represented as a string of its alphabet Σ. The full definition of the context free grammar will not be stated here, but can be found in [5]. Figure 3.3 provides an example of a derivation tree of the CFG, along with its string and process model interpretation.

In its implementation of the CFG, which is publicly available, PLG defines a set of basic *patterns*, or *production rules*, that are recursively applied to the process model. While the work of Burattin does not explicitly define these production patterns, they

will be defined in this work, since it is simpler to refer to these production patterns instead of the CFG when discussing the framework. These basic patterns can be seen in figure 3.4.

Some of the basic patterns of figure 3.4 contain an element called p_{empty}. This element can be thought of as the equivalent of a non-terminal symbol in the CFG, and can be replaced by any of the other basic patterns, including the containing pattern itself. For this reason, this element will be referred to as a *placeholder* of the other patterns. Since it is the equivalent of a non-terminal symbol, all finished generated process models contain no such placeholders.

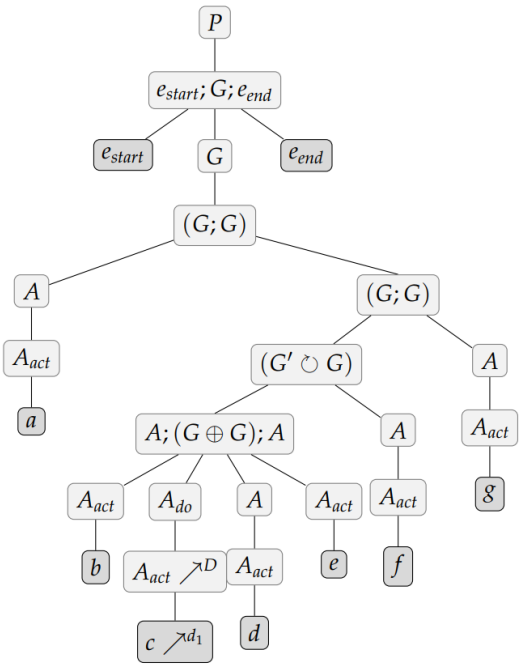
The generation framework of PLG allows the user to specify weights for each of the production rules in figure 3.4. The generation of a process model starts by defining a start and end event, such that the initial model is defined by the string "e_{start} ; p_{empty} ; e_{end}". Then, the rest of the model is generated by recursively replacing the placeholder p_{empty} with one of the basic patterns, which then replaces all of its own placeholders with other basic patterns, and so on, until there are no more placeholders left in the model, or until a specified maximum limit on the size of the process model is reached. What patterns are chosen to replace each placeholder is based on the weights assigned to each of the patterns at the beginning of the generation, which translates into the probabilities of choosing each pattern. These probabilities stay the same throughout the generation of each process model.

The extension proposed by this thesis mainly affects the continuous recalculation of the probabilities for choosing each pattern for each new pattern that is chosen, and as such shares most of the rest of the framework with PLG. Therefore, the commonalities of the PLG and the extension proposed in this thesis will be referred to as the *framework* of the proposed algorithm. For example, the basic patterns and way of recursively generating the algorithm is part of the framework.

The grammar used by the framework only has a limited expressiveness in terms of the possible business processes it can produce, and limits the framework to only producing block-structured processes, a term which is further explained in [26]. Because of this, the processes produced by the framework will all be sound processes, which for example means it does not contain any deadlocks. This soundness leads to the framework being unable to produce models containing any obvious flow-related errors that can be detected without domain knowledge[38].

Table 3.3: The metrics defined in [33], which are the focus of this work. See also table 3.2.

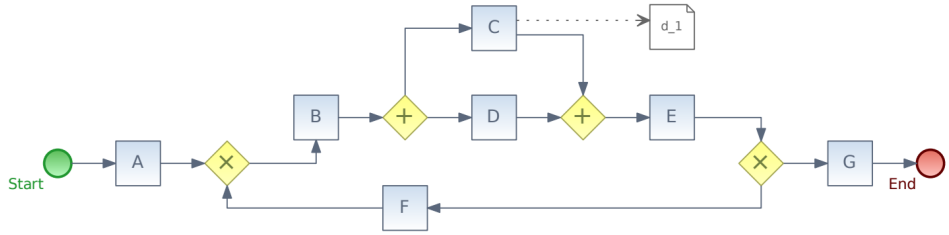
Category	Metric	Definition
Partitionability	Separability	$\Pi(G) = \frac{\lvert\{n \in N \mid n \text{is cut-vertex}\}\rvert}{\lvert N\rvert - 2}$ The ratio of cut-vertices to the number of nodes. A cut-vertex is a vertex that creates more separated components in a graph when removed.
	Sequentiality	$\Xi(G) = \frac{\lvert A\rvert \cap (T \times T)}{\lvert A\rvert}$ The ratio of arcs between non-gateway nodes to all arcs
	Structuredness	$\Phi_N = 1 - \frac{S_N(G')}{S_N(G)}$ The ratio of nodes in a subgraph G' to the nodes in G, where G' is obtained by applying a set of reduction rules defined in [33] as many times as possible. The reduction rules reduces structured blocks of a graph.
	Depth	$\Lambda(G) = max\{\lambda(n) \mid n \in N\}$The maximum depth of all nodes, where the depth of a single node is the number of split-gateways that have to be visited in order to reach it from the start node.
Connector interplay	Connector mismatch	$MM(G) = MM_{xor} + MM_{and}$ The number of gateway splits that are not matched to a gateway join.
	Connector heterogeneity	$CH(G) = -\sum_{l \in \{and,xor\}} p(l) * \log_2(p(l))$, where $p(l) = \frac{\lvert C_l\rvert}{\lvert C\rvert}$ The entropy over different gateway types.
	Control flow complexity	$CFC(G) = \sum_{c \in C_{and}} 1 + \sum_{c \in C_{xor}} d_{out}(c_{xor}) + \sum_{c \in C_{or}} 2^{d_{out}(c_{xor})} - 1$ Defined in [9] as a measure of complexity in terms of how many states of execution the designer of a process has to keep in mind
Cyclicity	Number of cycles	$CYC_N = \lvert L\rvert$ States the number of distinct cycles in G. This is not a metric defined in [33] as other metrics, but is rather defined by Nissen in [35].
Concurrency	Token split	$TS(G) = \sum_{c \in C_{or} \cup C_{and}} d_{out}(n) - 1$ The output-degree of AND-joins and OR-joins minus one.



(a) An example of a derivation tree from the CFG of PLG

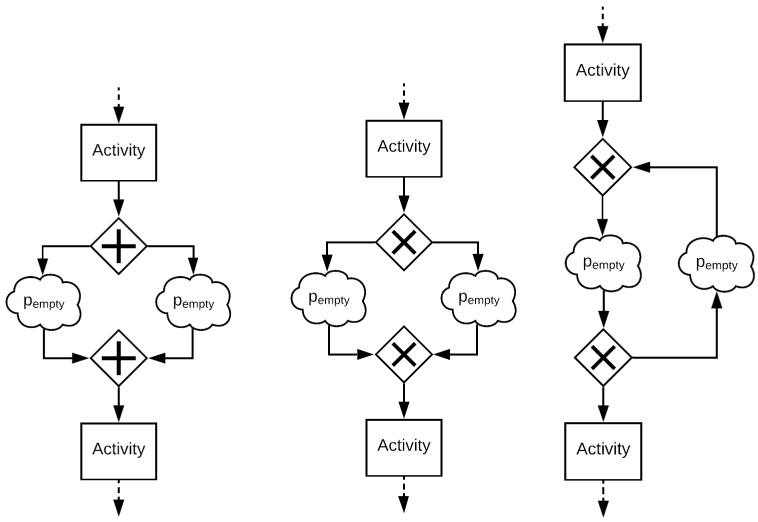
$$e_{start}; \left(a; \left(\left(b; \left(c \nearrow^{d_1} \oplus d\right); e \circlearrowleft f\right); g\right)\right); e_{end}$$

(b) The string derived from 3.3(a)



(c) The BPMN representation of 3.3(a)

Figure 3.3: An example of a process model produced by the CFG of PLG. Reproduced from [5].



(a) *Parallel execution* pattern

(b) *Mutual exclusion* pattern

(c) *Loop* pattern

(d) *Activity* pattern

(e) *Skip* pattern

Figure 3.4: The basic patterns, or production rules, of PLG, which are derived from its CFG. The interpretation of the skip pattern can be seen in figure 3.5.

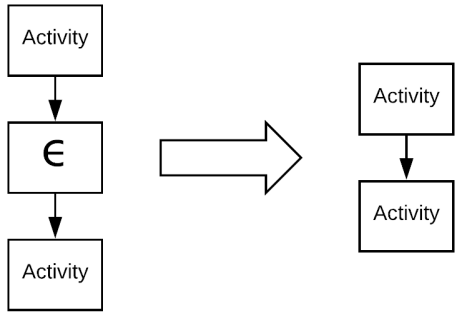


Figure 3.5: The interpretation of the skip pattern of 3.4(e). Its equivalent string representation would be "A_{act} ; ϵ ; A_{act}" $\rightarrow$ "A_{act} ; A_{act}".

CHAPTER 4

Process Model Generation Algorithm

When constructing a benchmark of process models, the desired properties of the models can vary based on the algorithm being benchmarked, and the purpose of the benchmark. This means that putting researchers in direct control of what properties the generated process model collections will have is important. This is why doing this, in the form of the metrics of the process model collection, is the main focus of this work. However, this work will also assume that, in general, a good benchmark is *diverse*, i.e. that its characteristics are diverse for those that are not directly specified by the researchers. The logic behind this is that while a generation algorithm should ensure that the process models have the specified characteristics, it should also try to make sure that the benchmark covers as many cases as possible, so the benchmark has an increased chance of covering all cases the algorithm will have to deal with in real life. In continuation of this, it will also be assumed that a good benchmark is *realistic*, i.e. contains a realistic collection of process models, and the model generation algorithm should also try to facilitate this.

The main idea of the generation algorithm is that instead of specifying the probabilities of each possible pattern to generate, as is the case for existing generation algorithms, users specify the characteristics they want their process model collection to exhibit. The algorithm tries to achieve these characteristics by dynamically assigning and reassigning the probability of each pattern at run-time, based on the current and desired state of the model. This section explains the algorithm in further detail, of which a pseudo-code description can be seen in algorithm 1 through 8.

Note that the algorithm proposed in this work only deals with the generation of the structural properties of process models. This means that it does not concern itself with the names of components, or the production or consumption of data objects, as is the case with the PLG framework. It only concerns itself with the generation of models of basic workflow patterns.

The characteristics that the algorithm takes as input from the user, asides from the size of the collection, are passed as pairs of a *metric* and the *distribution* of its target values. Each pair specifies a characteristic of the collection of generated process models, and will be referred to as the *obligation* for that metric. As such,

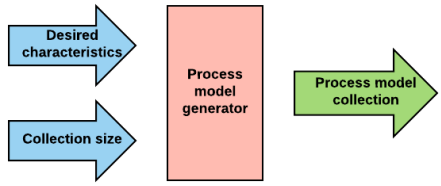


Figure 4.1: The inputs and outputs of the generation algorithm.

each obligation is a pair of a metric and a specific distribution (with a specified mean, variance, etc.). Since the obligation specifies the desired distribution of each metric, this means that the metrics of the whole process model collection will follow this probability distribution. The distribution of the generated collection is discussed in further depth in section 4.2.

While the algorithm takes as input a metric and the distribution of its target values, each process model is generated from a single *target value*, which is obtained from the specified distribution. This value dictates the value that the specific process model should be generated with. Each of these values are generated at the beginning of the generation of the process model, based on the obligation and the specified distribution, in line 5 in algorithm 2. The pair of a metric and its target value is simply referred to as a *target* for the process model.

The generation algorithm for each process instance is based on the framework of PLG, which has been explained in section 3.3. This means that, just as PLG, the algorithm generates each process model by recursively generating new patterns based on the CFG, until no more patterns can be generated. At each recursive generation, the algorithm decides which new pattern or component to generate, based on the weight that is assigned to each. However, while the generator of PLG assigns the weights for each pattern at the beginning of the process generation, after which they remain unchanged for the rest of the generation, the algorithm proposed by this work dynamically assigns new weights to each pattern at each recursive iteration.

The way the weights are distributed is decided by the current state of the generated model, i.e. what the model currently looks like, as well as what the model is supposed to look like, i.e. its metric target values. This means that at each recursive generation step, the program looks at the metrics of the currently generated model, and compares this to the target value. Based on the difference, the algorithm decides to which degree the generation of a certain pattern is beneficial, and assigns a weight to the pattern accordingly. The exact procedure for assigning weights to each pattern, and what makes it beneficial, is a bit more complex than this, and is discussed in further detail in section 4.3.

4.1 Pseudo-code Description

Algorithms 1 through 8 describe the generation algorithm proposed by this work in terms of pseudo-code. The workings of the algorithm is the same as what is described elsewhere in the thesis, but is meant both as a more precise and concise description of it. The `GENERATE-PROCESS-MODELS` function should be considered the external interface to the algorithm, such that it takes inputs as described by its parameters and produces an output in accordance with the function's return statement, as seen in figure 4.1. The framework of the generation engine of PLG can be recognized as algorithms 1, 2 and 3, which are largely equivalent. There are however a few differences, the most significant being in line 7 of algorithm 3, where the placeholder p_{empty} in the parent pattern, i.e. the pattern that is to be replaced by the chosen pattern, is replaced by the chosen pattern with placeholders, before these are replaced themselves. The reason for this difference is discussed in section 4.3.6. Another minor difference is in line 5 of algorithm 2, where the target value for each metric is determined, as well as the parameters and return-values of the functions. However, the effects of the latter, minor differences are not seen before `COMPUTE-WEIGHTS`, which is where the metric target values are used. This means that the main differences between the frameworks can be found in `COMPUTE-WEIGHTS`, and is in accordance with the main contributions from this work being the way that probabilities for each pattern are dynamically determined at each iteration based on the current and target state of the model. As such, all other parts of the algorithm that are not contained within this function will often be referred to as the *framework* of the algorithm, since it shares this with PLG.

The pseudo-code largely follows the conventions of [11], which for example means that objects passed as arguments, including arrays, are passed by reference. Certain deviations from the convention, as well as other conventions, are as follows.

- Assignments to variables are expressed as $\leftarrow$, such as $x \leftarrow y$, instead of $x = y$.
- Strings are described in a java-like syntax, where for example "Hello" + "World" would produce the string "HelloWorld".
- p_{empty} is regarded as an object. This means that passing a p_{empty}-object in `GENERATE-PATTERN` passes the corresponding object, not simply the empty string ϵ.
- The patterns in the pseudo-code, such as $p \in P$ in line 4 of algorithm 1, are as defined as in figure 3.4.
- Certain functions called are not formally described as pseudo-code, as their inner workings are not essential to the contributions of this work. The functions in question are as follows.
  - `choosePattern`: Chooses a pattern based on the weights assigned to them. The probability of choosing each pattern is the weight assigned to the

pattern, divided by the total number of weights of all patterns. In other words, the probability of p, $prob(p) = W(p)/\sum_{p \in P} W(p)$, where p, P and W are as defined in lines 4 and 5 in the code.

- `random(f)`: Chooses a random number based on a specified distribution f, as defined in line 3 of algorithm 1. This could be any distribution, but is intended to be one that reflects a realistic distribution of the specified metric, as discussed in section 4.2. For the purposes of this work, a Poisson distribution has been chosen, which means that the function returns a random value following a Poisson distribution with the specified mean value, the implementation of which is discussed in section 5.0.2.
- `removePlaceholders(`g`)`: This is a method for removing the placeholders p_{empty} that exist in the process model g. The scheme for doing so is explained in section 4.3.6. The method returns the resulting model.
- `isLargerThanAreaOfComfort(`n_p`)`: This method returns `true` if the number of placeholders in the generated process model n_p is *higher* than the largest value in the range of values defined as the *area of comfort* for n_p in section 4.3.3.
- `isSmallerThanAreaOfComfort(`n_p`)`: This method returns `true` if the number of placeholders in the generated process model n_p is *lower* than the lowest value in the range of values defined as the *area of comfort* for n_p in section 4.3.3.
- `reachedTarget(`m`, `t`)`: This function compares the current value m of a metric to its target value t. When comparing, a margin e is introduced, such that the function returns the boolean value of $t - e < m < t + e$. This error margin e has been set to 5 % of t.

4.2 Probability Distribution of the Metrics

While it is difficult to define the characteristics of what constitute *realistic* process models, or what process models resemble real-life ones, it is possible to say something about the probability distribution of real process model collections. The work of Mendling in [33] analyzes four real-life collections of process models in terms of a range of metrics and the mean and variance of their probability distribution functions. From this work, we can see that even though each metric has varying mean and variance, each variance $\sigma > 0$, and the mean and variance tend to fall within a factor 2 of each other. While not conclusive, this indicates that collections of real process models tend to follow a non-uniform probability distribution function. This is also indicated in [40], where the distribution of the length of the longest path for a realistic collection is displayed as a histogram which clearly indicates a curved, Gaussian-like distribution.

It is however difficult to say anything definite about the kind of probability distribution function real process model collections tend to follow, without an extensive

Algorithm 1 GENERATE-PROCESS-MODELS

inputs:
1: $O \leftarrow \{(m, f) | m \in M, f \in F\}$ ▷ The obligations of the algorithm, such that $O(m) = d$ is the probability distribution for m
2: $s \in \mathbb{N}$ ▷ The size of the generated collection

globally defined constants:
3: F ▷ all possible probability distributions
4: P ▷ all possible patterns
5: $M : \text{string} \rightarrow \mathbb{R}$ ▷ all possible metrics
6: $M' \leftarrow \{m | (m, f) \in O\}$ ▷ the set of metrics passed as obligations in O

7: **function** GENERATE-PROCESS-MODELS(O, s)
8: G ▷ the collection of generated process models
9: **for** 1 to s **do**
10: $G \leftarrow G \cup \{\text{GENERATE-PROCESS-MODEL}(O)\}$
11: **end for**
12: **return** G
13: **end function**

Algorithm 2 GENERATE-PROCESS-MODEL

inputs: ▷ Global variables defined in algorithm 1 are also accessible here
1: O ▷ The set of obligations, as defined in algorithm 1

2: **function** GENERATE-PROCESS-MODEL(O)
3: $T : M \rightarrow \mathbb{R}$ ▷ The metric target values, such that $T(m)$ is the target value of m
4: **for all** $m \in M'$ **do**
5: $T(m) \leftarrow \text{RANDOM}(O(m))$
6: **end for**
7: $g \leftarrow$ "e_{start} ; " + p_{empty} + " ; e_{end}" ▷ the generated process model
8: **replace** $p_{empty} \in g$ with GENERATE-PATTERN(T, g, p_{empty})
9: **return** g
10: **end function**

analysis of these collections. In order to be as general as possible, while still trying to achieve a closeness to real process model collections, the generation algorithm proposed by this work supports, in principle, *any probability distribution function* for the metrics of the generated collection. It is however worth noting that for the implementation of the algorithm to support a certain probability distribution, a random number generator as specified in line 5 of algorithm 2 is required.

Algorithm 3 GENERATE-PATTERN

inputs: ▷ Global variables defined in algorithm 1 are also accessible here
1: T ▷ The set of metrics and target values, as defined in line 3 of algorithm 1
2: g ▷ The currently generated process model
3: $p_{location} \in g$ ▷ The placeholder pattern that is to be replaced

4: **function** GENERATE-PATTERN(T, g, $p_{location}$) ▷ Recursively generates a pattern in place of $p_{location}$ in g
5: $W \leftarrow$ COMPUTE-WEIGHTS$(T, g, p_{location})$
6: $p_{chosen} \leftarrow$ CHOOSEPATTERN(W)
7: **replace** $p_{location}$ in g with p_{chosen}
8: **for all** $p_{empty} \in p_{chosen}$ **do**
9: **replace** p_{empty} with GENERATE-PATTERN(T, g, p_{empty})
10: **end for**
11: **return** p_{chosen}
12: **end function**

4.3 Weighing Patterns

The weighing of patterns, or computing their weights, is done in `COMPUTE-WEIGHTS` at line 4 in the pseudo-code. The main idea of assigning weights is that each target contributes with a certain value to each pattern's weight, depending on the benefit of producing the pattern in relation to the metric in question. This means that the final weight of each pattern is the sum of the weights provided from all targets. As an example, consider a generation instance with one target value for *number of activities*, and another for *number of gateways*. When calculating the weight for a sequence pattern, it might receives a weight from the number of activities of 0.5. When it then receives a contribution of 0.25 from the number of gateways, this will add up to a total weight of $0.5 + 0.25 = 0.75$.

The total contribution of weights from a target to all patterns is always 1, meaning that each target has a total weight of 1 that it can split up and assigned to different patterns, depending on how beneficial it finds each one. What is considered beneficial for a particular target follows a set of rules, and is discussed in section 4.3.3. In order to assure that the total sum from each pattern is 1, each weight the target wants to give out is normalized with respect to the sum of all weights. This is described in algorithm 6 in the pseudo-code. The weight given out from each target to a pattern can be either positive or negative. In the case that it is negative, its absolute value will count towards the total value it is allowed to give out, meaning that contributing with -0.5 and 0.5 to a pattern would count equally towards giving out a total value of 1.

By giving each target a static value of 1 that it can assign in total, it is ensured that each target contributes equally to what pattern is chosen, so as to restrain each

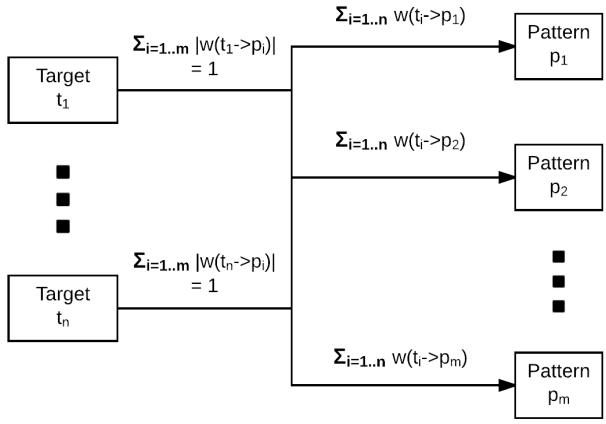


Figure 4.2: The scheme of assigning weights from each target to each pattern. Notice that the sum of all weights each target gives out, $\sum_{i=1}^{m} |w(t_n \rightarrow p_i)|$, is equal to 1. Each pattern receives a part of this total weight from each target, which equates to the total weight assigned to the pattern.

target. If this were not the case, an overly active target might assign either very high or low weights to each pattern, resulting in a high sum of all weight contributions given out. Meanwhile, another target might assign a weight to only one pattern, giving it a low total weight contribution. When compared to the overly active target, its contribution would easily drown among all the other contributions, so the overly active target would dictate what patterns are generated.

4.3.1 Interdependence of Targets

When considering a metrics-based generation algorithm, the statistical independence of the targets and their metrics are important. When two or more targets of the generation algorithm somehow have an effect on each other during generation, the algorithm might not produce optimal results, as the desired value of one target might conflict with another. In order to discuss this concept, this thesis defines the notion of *interdependence of targets*. In general, if two targets are interdependent, it means that they might somehow affect one another during the generation. The concept of interdependence is both reliant on the metric of the target, as well as the basic patterns of the generation framework used. Conversely, if two targets are not interdependent, a model could always be generated with any combination of values for the metrics, since they would never conflict.

Algorithm 4 COMPUTE-WEIGHTS

inputs: ▷ Global variables defined in algorithm 1 are also accessible here
1: T ▷ The set of metrics and target values, as defined in line 3 of algorithm 1
2: g ▷ The currently generated process model
3: $p_{location} \in g$ ▷ The pattern in g where p would be inserted

4: **function** COMPUTE-WEIGHTS(T, g, $p_{location}$) ▷ Calculates the total weights assigned to each pattern based on T and the state of g
5: $W : P \to \mathbb{R}$ ▷ the set of weights for all patterns, such that $W(p)$ is the weight of pattern p
6: **for all** $p \in P$ **do**
7: $W(p) \leftarrow \sum_{m \in M}$ COMPUTE-WEIGHT$(T, m, p, g, p_{location})$
8: **end for**
9: **return** W
10: **end function**

Algorithm 5 COMPUTE-WEIGHT

inputs: ▷ Global variables defined in algorithm 1 are also accessible here
1: T ▷ The set of metrics and target values, as defined in line 3 of algorithm 1
2: m ▷ The metric of the target value that is giving out the weight
3: p ▷ The pattern that the weight is going to be assigned to
4: g ▷ The currently generated process model
5: $p_{location} \in g$ ▷ The pattern in g where p would be inserted

6: **function** COMPUTE-WEIGHT(T, m, p, g, $p_{location}$) ▷ calculates the weight given to p based on m, $T(m)$, $m(g)$, if p is inserted in g at $p_{location}$
7: $b_m \leftarrow$ CALCULATE-BENEFIT$(T, m, p, g, p_{location})$
8: $b_M \leftarrow \sum_{p \in P}$ CALCULATE-BENEFIT$(T, m, p, g, p_{location})$
9: $w_m \leftarrow b_m / b_M$ ▷ the normalization of b_m
10: **return** w_m
11: **end function**

In this work, two kinds of interdependence will be defined. The first one of these will be called *inherent interdependence*, which means that the definition itself of each metric of two targets makes them dependent, because they partly contain the same measures. A simple example is the number of AND-splits, G_{AND}, and the total number of gateway splits, G_{ALL}, in a model. If a model can contain both AND- and OR-gates, this means that $G_{ALL} = G_{AND} + G_{OR}$. Thus, an increase in G_{AND} will also lead to an increase of G_{ALL}, which would make their targets inherently interdependent.

Another case is what will be defined in this work as *non-inherent interdependence*,

which, as the name suggests, is not inherent to the measures that make up metrics themselves, but is rather due to the basic patterns used by the generation framework, in this case PLG. Imagine if a generation algorithm can choose between one of two patterns - creating an activity, or creating an activity *and* an AND-split. Now let us say that the generation algorithm has been set to generate a model with 10 AND-splits, but only 5 activities. The number of AND-splits and activities in a model are not inherently dependent on each other, since adding an activity to the model does not necessarily add another AND-split, and vice versa. However, with only the two aforementioned generation patterns available, generating such a model would be impossible, since the algorithm would always generate one activity each time it created an AND-split. In other words, even though the metrics are not inherently dependent on each other, they are dependent on each other with respect to the possible pattern, or non-inherently interdependent. This is actually the case for the generation framework of PLG, where generating a parallel execution pattern also generates two activities. Conversely, if the pattern generating an AND-split did not also produce an activity, it would indeed be possible to create such a model with twice as many gateways as activities. The generation of a pattern affecting one metric would not affect the other, and thus the metrics would not be interdependent.

While the above is a trivial example, the same situation can in principle occur each time the generation of a certain pattern affects more than one metric at a time. For example, the creation of a loop would affect the cyclicity of the model, but it would also affect its control flow complexity, as it involves adding a pair of gateways, thus introducing more possible states to the model. It is arguable that the independence of metrics is lowest when the metrics are within the same category, or involves the same measures, in which case we might have both inherent and non-inherent interdependence. Note that two metrics can correlate when looking at their values for a collection of models, while still having neither inherent nor non-inherent interdependence.

4.3.2 Monotonicity of Metrics

This thesis will make a distinction between monotonic and non-monotonic metrics, with regards to the patterns used by the framework. This is a term defined by this work, and draws inspiration from the notion in mathematics of monotonic functions. If a metric is monotonic, it is either non-increasing or non-decreasing during the generation of the model, when using a certain set of production patterns. Using the number of activities as an example, this will always be monotonically increasing during generation, as no patterns in the framework will ever remove any activity.

When a metric is non-decreasing with respect to a set of patterns, it will also be referred to as a *monotonically increasing metric*. Likewise, monotonic metrics can also be monotonically decreasing, which means that it can never increase with respect to the set of production patterns. On the other hand, non-monotonic metrics might both increase or decrease during generation, as the addition of different patterns may

increase or decrease them. Non-monotonic metrics have the ability to enter into *local maxima*. This can for example happen if a metric is being generated towards a target value, but in order to increase from its current value to the target value, it first has to decrease. As discussed in section 6.1, this can prove a challenge to the algorithm, and is therefore important to note here.

4.3.3 Benefit of Patterns

The input to the benefit calculation in algorithm 6 is five-fold. First of all, it needs to know which combination of metric and pattern the benefit is being calculated for, in the form of m and p. It also needs to know the state of the currently generated model, so g is passed, as well as which pattern the generated pattern would replace, as g and $p_{location}$. Along with the target value of the metric t, it can use this to calculate how much closer to the target value generating the pattern in question gets the model, as Δd in line 10 of algorithm 7.

The goal for the rules that govern the benefit of a pattern is twofold. Firstly, it should make sure that the target value is reached. Second, it also has to make sure that the generation of the model terminates at the right time, meaning that when the target value is reached, or a better solution can not be reached, the algorithm should stop. If not, it should continue. To facilitate this, the rules governing the benefit has been split in two, where one part states the benefit of choosing the pattern in terms of distance from the current value to the target value, and another for the benefit of choosing the patterns in terms of termination of the generation. These parts will be called the *metric benefit* and *termination benefit*, respectively.

The idea is that normally, the metric benefit of the pattern should govern what pattern is chosen. The metric benefit is calculated simply as the difference in distance to the target value from generating a pattern. In other words, the metric benefit is given as metric benefit $= -\frac{\Delta d}{\text{target value}} = \frac{|d_{after}|-|d_{before}|}{\text{target value}}$, where d is the distance from the current value to the target value, and $d_{after} = |t - m_{after}|$ and $d_{before} = |t-m_{before}|$ are the absolute distances after and before the pattern would be generated in the current model, respectively, where t is the target value of metric m. This means that the closer a pattern will bring a metric to its target value, the more desirable it is, as the metric benefit will be higher. If Δd is positive, i.e. the distance is increasing, the benefit will be negative, i.e. it is not desirable. The definition of the metric benefit is calculated in algorithm 7.

However, there are certain cases where we also have wishes for the current growth potential of the algorithm, which is governed by the number of placeholders p_{empty} in the model. These cases can occur when the specific metric has reached its target value, or is within the threshold of it, defined as *5 %* of the target value, in which case it is beneficial for the metric to terminate generation of more patterns that might potentially bring the model further away from the target value. In such cases, the termination benefit will return a high benefit for patterns that decrease the number of placeholders in the model. Also, an *area of comfort* has been defined for the number

of placeholders in a model, which is a range for this number that the algorithm should be within during the generation algorithm. This range is defined as between 3 and 5 placeholders, and ensures that the algorithm does not get too many placeholders, such that it becomes harder to stop when target values are reached, and so that it will not inadvertently terminate while the target values have not yet been reached. This is also described in algorithm 8 of the pseudo-code.

The termination and metric benefits are combined into the total benefit, so that they have the same effect on the total benefit. This means that the algorithm assures that $|\text{metric benefit}| = |\text{termination benefit}|, \text{termination benefit} > 0$. This means that $\text{benefit} = \text{metric benefit} + \text{termination benefit}, \text{termination benefit} > 0$ and $\text{benefit} = \text{metric benefit}, \text{termination benefit} = 0$. Note that this benefit is calculated for each combination of metric and pattern, and is the same value as returned by algorithm 6.

Algorithm 6 CALCULATE-BENEFIT

inputs: ▷ Global variables defined in algorithm 1 are also accessible here
1: T ▷ The set of metrics and target values, as defined in line 3 of algorithm 1
2: m ▷ The metric of the target value $T(m)$ that the benefit is calculated for
3: p ▷ The pattern that the benefit is calculated from
4: g ▷ The currently generated process model
5: $p_{location} \in g$ ▷ The pattern in g where p would be inserted

6: **function** CALCULATE-BENEFIT(T, m, p, g, $p_{location}$) ▷ calculates the benefit for $T(m)$ of generating p in g in the place of $p_{location}$
7: $b_m \leftarrow$ CALCULATE-METRIC-BENEFIT$(m, T(m), p, g, p_{location})$ ▷ The metric benefit
8: $b_t \leftarrow$ CALCULATE-TERMINATION-BENEFIT$(m, T(m), p, g)$ ▷ The termination benefit
9: **if** $|b_t| > 0$ **then** ▷ If the termination benefit is not zero, both the metric and termination benefit matter equally
10: $b_t \leftarrow b_t * \frac{|b_m|}{|b_t|}$ ▷ Ensure that b_t has the same magnitude as b_m
11: **return** $b_m + b_t$
12: **else**
13: **return** b_m
14: **end if**
15: **end function**

4.3.4 Optimizing the Model State

The idea of calculating the benefit of a pattern based on how much closer it gets the metric to a target value means that the algorithm can be seen as an optimization

Algorithm 7 CALCULATE-METRIC-BENEFIT. See section 4.1 for a description of `removePlaceholders()`

inputs:
1: m ▷ The metric that the benefit is calculated for
2: t ▷ The target value for m
3: p ▷ The pattern that the benefit is calculated for
4: g ▷ The currently generated process model
5: $p_{location} \in g$ ▷ The pattern in g where p would be inserted

6: **function** CALCULATE-METRIC-BENEFIT(m, t, p, g, $p_{location}$) ▷ Calculates the metric benefit for m of generating p in g
7: $d_{before} \leftarrow |t - m(\text{REMOVEPLACEHOLDERS}(g))|$ ▷ Distance (numerical difference) to target value before p is inserted in g
8: g_p ▷ g with p inserted in place of $p_{location}$
9: $d_{after} \leftarrow |t - m(\text{REMOVEPLACEHOLDERS}(g_p))|$ ▷ Distance (numerical difference) to target value after p is inserted in g
10: $\Delta d \leftarrow d_{after} - d_{before}$
11: **return** $-\frac{\Delta d}{t}$
12: **end function**

problem, which is about reducing the relative distance between the actual values of the metrics and their target values, i.e. $|\frac{\Delta d}{t}|$, for each metric. The algorithm can be seen as an approximation of the optimization problem for $-\frac{\Delta d}{t}$ in line 11 of algorithm 7, in which the algorithm should minimize the absolute average value for all metrics for $|\frac{\Delta d}{t}|$, or minimize the euclidean length of the vector obtained by combining $-\frac{\Delta d}{t}$ for all metrics. While just an approximation, since this value is not directly used, and the termination benefit can also play a role, it illustrates the basic idea of the algorithm well. It also allows us to talk of a model state, the improvement of which happens as $|-\frac{\Delta d}{t}|$ for all metrics decreases. When the model state can not be improved any longer, the model state is optimized.

In principle, there are multiple ways of calculating this improvement of the model state, by choosing each pattern. For certain combinations of patterns and metrics the difference in distance is always the same, regardless of the current state of the model. For example, producing a mutual exclusion pattern will always contribute 2 to the number of AND-gateways, no matter how many other AND-gates exist in the model. In such cases, it would be possible to pre-program Δd, such that it would only have to be looked up at run-time. This is however not possible for certain other metric-pattern combination. One example is the diameter, which would increase if the chosen pattern was located on the longest path, given that the pattern has a length $l > 0$. However, if the same pattern was generated on a shorter path, this would not necessarily be the case. In other words, the change in difference depends on the current state of the model. Another common case where the difference in distance can

Algorithm 8 CALCULATE-TERMINATION-BENEFIT. See section 4.1 for a description of the methods `isLargerThanAreaOfComfort()` and `reachedTarget()`

inputs:
1: m ▷ The metric that the benefit is calculated for
2: t ▷ The target value for m
3: p ▷ The pattern that the benefit is calculated for
4: g ▷ The currently generated process model

5: **function** CALCULATE-TERMINATION-BENEFIT(m, t, p, g) ▷ calculates the termination benefit for m of generating p in g
6: n_p ▷ Number of placeholders p_{empty} in g
7: Δn ▷ difference in placeholders after introducing p in g
8: shouldDecreasePlaceholders = isLargerThanAreaOfComfort(n_p) OR reachedTarget(m, t) ▷ Whether the number of p_{empty} in g should decrease
9: shouldIncreasePlaceholders = isSmallerThanAreaOfComfort(n_p) AND !reachedTarget(m, t) ▷ Whether the number of p_{empty} in g should increase
10: **if** shouldDecreasePlaceholders **then**
11: **if** $\Delta n < 0$ **then**
12: **return** 1
13: **else**
14: **return** -1
15: **end if**
16: **else if** shouldIncreasePlaceholders **then**
17: **if** $\Delta n > 0$ **then**
18: **return** 1
19: **else**
20: **return** -1
21: **end if**
22: **else**
23: **return** 0
24: **end if**
25: **end function**

not be predetermined is with certain metrics, which are determined as a ratio. Here is the case that the magnitude of the difference, and whether it is positive, negative, or zero, is dependent on the values of the numerator and denominator before the pattern is added. An example of such a metric is connectedness, where a sequence pattern contributes more when there are fewer edges, than when there are many. Another example is the coefficient of connectivity, given as the ratio of connectors to nodes, which will always go towards the value 1 as subsequent sequence patterns with an activity are added to the model. If this happens as the first thing after the start and end nodes, the value will increase towards 1 from 0.5, and if there has been created a

large number of gateway patterns, such that the metric is higher than 1, generating a sequence pattern with an activity would decrease the value of the metric.

For the above cases, another way of determining the difference in distance to the target value is needed. In this work, the difference in distance to the target value Δd is calculated by simulating the generation of the pattern on the current model, and calculating the new value of the metric, to compare it to the initial value value. This process can be seen in algorithm 7. This method was chosen because of its generality and simplicity, such that all pattern-metric combinations can be supported. Notice that the simulation is reliant on knowing the current state of the model by its metrics, which is explained in further detail in section 4.3.5.

An alternative to determining Δd by simulation that was considered was to create specific predetermined rules for determining the difference in the case of each metric, based on a set of rules on the current state of the model. While such a solution might achieve a better running time for complex metrics, since the difference would only have to be looked up, the difficulty of implementing such rules, especially for ratio-based metrics, was considered high as compared to the difficulty of adapting the framework to support the run-time evaluation of metrics, as described in section 4.3.6. Also, requiring the implementation of rules specific to each supported metric would also mean that an extra overhead in the implementation of new metrics would be introduced, which is undesirable.

4.3.5 Evaluation of Process Model Metrics

During a process generation instance, the algorithm continuously recalculates the probabilities of each pattern, based on the current state of the model, compared to its desired state. Since a model's state is defined by its metrics in the algorithm, this means that it measures the metrics of the current model, and compares this to the desired value for the metrics, in order to decide what pattern to generate next. This means that the generator needs a way of automatically calculating the metrics of process models at run-time. This is done as a part of the calculation of the benefit of generating a pattern, in lines 7 and 9 of algorithm 7.

Since the generator is heavily dependent on a metrics calculator, as it has to re-measure all metrics at every recursive iteration, it is very dependent on the calculator's qualities, specifically its time complexity and supported metrics. As it has to be run at every iteration, its direct impact on the time complexity is evident, and will be discussed further in section 4.4. When it comes to the supported metrics of the calculator, since those supported by the algorithm need to be calculated at every iteration, the algorithm can only support those metrics that are also supported by the calculator. The supported metrics of the algorithm can be seen in figure 5.1 and 5.2.

4.3.6 Supporting Run-time Evaluation

When metrics are evaluated at run-time, this is done on the current state of the generated process models. However, in order to do this on the "unfinished" process model at each iteration, certain modifications to the PLG generation framework are needed.

When the PLG framework recursively calls its `GENERATE-PATTERN`-equivalent, it waits until the recursive call returns to insert the generated pattern into its chosen pattern p_{chosen}, and connect it to the rest of the pattern. However, this means that at each call of `GENERATE-PATTERN`, the generated pattern is not connected to any other parts of the model, until control returns to its caller. In other words, the framework generates model components in a top-down fashion, and connects them in a bottom-up fashion. If a metric is calculated for the process model, the components would not be connected, such that there is no information about the control flow of the model, which is necessary for metrics such as the diameter. This is shown in figure 4.3, in which the components of each pattern are generated on the way "down" in the recursion, while the same components are not connected until the same call returns back "up" the recursion.

In order to solve this problem, a framework that both generates *and* connects models in a top-down fashion is needed. To achieve this, the algorithm generates a placeholder, p_{empty}, in each `GENERATE-PATTERN`, which it inserts into the generated model in place of the patterns that are to be generated by subsequent recursive `GENERATE-PATTERN`-calls, in line 9. This means that calls further down the recursion will have accurate information about the flow state of the model, and can properly evaluate metrics from this state. This new scheme is shown in figure 4.4, in which both the generation and connection of components happen on the way down the recursion, i.e. before consecutive recursive calls are made.

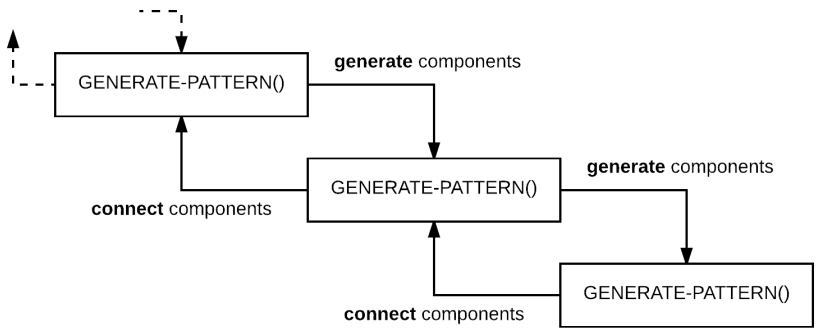


Figure 4.3: Scheme for generating patterns in the PLG framework.

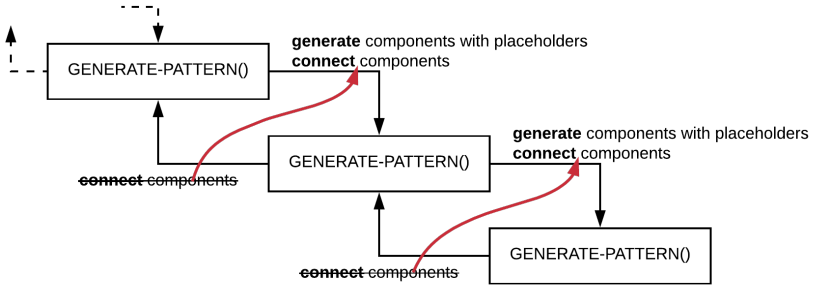


Figure 4.4: The new scheme for generating patterns, which allows for run-time evaluation of metrics on unfinished models.

Generating placeholder to achieve a top-down connection of process model components introduces another problem, which is how to handle the placeholder itself when evaluating a metric. Since the placeholder represents a pattern that *is to be* generated, but has not been so yet, it is in fact an empty pattern, which is equivalent to a skip pattern. This is also reflected by the string representation of the placeholder as $p_{empty} = \epsilon$, which is the same as a skip pattern. This means that in order to support the run-time evaluation of metrics, all placeholders in the currently generated model are treated as empty strings, such that "p_{before} ; " + p_{empty} + " ; p_{after}" = "p_{before} ; ϵ ; p_{after}" = "p_{before} ; p_{after}".

The time complexity of evaluating each metric depends on the metric itself. Certain metrics are fairly simple, and can be computed in constant or linear time. One such is the number of gateways, which involves counting the number of gateways in the process model, which can be done in linear or constant time, depending on the data structure that represents the process model. Other metrics are harder to calculate, such as cyclicity, which involves solving the problem of finding all cycles of a directed graph, which has several research papers discussing possible solutions, such as [25, 37], which all take significant time to compute, i.e. more than proportional to the size of the graph. As section 4.4 describes, the time that compute each metric takes can have a big impact on the total running time of the algorithm, especially when it is complex. A figure of the implemented metric calculators, and their running times can be seen in table 5.1 and 5.2.

4.4 Running time

While the running time of the algorithm is not a major area of focus for this thesis, the extension of the PLG generation framework proposed contains a significant amount

of computation. This section will present a brief analysis of the running time of the framework, and identify what is likely to be the most time consuming aspects. If the running time of the algorithm is to be improved, the identified areas are prominent candidates for inspection.

The original generation framework of PLG is very basic, and requires little computation. Assuming that the generated process models are generally of a certain size, significantly larger than the number of patterns, the most time-consuming part of the algorithm will likely be the number of recursive calls to its `GENERATE-PATTERN()` equivalent. While there are no guarantees, it is reasonable to assume that the number of calls to `GENERATE-PATTERN()` is proportional to the size s of the generated model in the end. This means that the original framework of PLG should have a running time of $O(s)$ for generating a single model, which means a running time of $O(n * s)$ when generating n models.

Instead of just using predetermined weights for each pattern, the extended framework computes these weights at each call of `GENERATE-PATTERN`, in line 5, which is quite computationally demanding. Basically, for each pattern, the algorithm computes the sum, for all specified metrics, of the normalized benefit of choosing the pattern, the latter of which requires computing the benefit of all other patterns, for that metric. Since the number of patterns are constant, this adds up to a running time of $O(|M|) * \text{timeof}(\texttt{BENEFIT()})$. Assuming that copying and converting the currently generated model g for simulation purposes in `removePlaceholders()` takes $O(s)$ time, where s is the size of the generated model, then the `CALCULATE-BENEFIT` function takes between s and however long the computation of the metrics of g takes. As discussed in section 4.3.6, the time it takes to compute the value of a metric is highly dependent on the metric itself, and could very well take less than, proportional to, or longer than time than s. Therefore, stating a running time for the algorithm is difficult, but making the assumption that calculating the most complex metric takes longer than converting g in line 9, the `COMPUTE-WEIGHTS` function takes $O(|M| * t_{metric}$ time, where t_{metric} is the time it takes to compute the most complex metric. This leads to a total running time of $O(n * s * |M|) * t_{metric}$, for which $t_{metric} >= O(s)$, or at least $O(n * s^2 * |M|)$.

Looking at the running time of the algorithm, if assuming that the set of basic patterns $|P|$ is significantly smaller than the average size of generated process models, then the terms that are going to be most significant to the running time are s and n. Since the contribution of this work only adds one term dependent on s, namely in $t_{metric} >= O(s)$, the time for computing metrics of the model have a very significant impact on the total running time.

This page has been intentionally left blank

CHAPTER 5

Software Implementation Details

This chapter describes the implementation of the algorithm described in chapter 4, which is used to obtain the results of chapter 6. Readers who are not interested in the details of the implementation of the algorithm should be able to skip this chapter, without losing out on information significant to the contribution of the work, with the exception of the list of implemented metric calculators of table 5.1 and 5.2.

The algorithm has been implemented as an extension to the PLG project, of which the source code is publicly available, as mentioned in [5]. As such, it is implemented as a Java project. It also readily supports exporting generated models as Petri nets using the PNML standard [20], as BPMN models, or as graphical representations of these using the Graphviz format [16]. It also supports a GUI for users, which allow users to specify their input as described in section 3.3, however, this is not supported for the extension proposed by this work, as will be described in 5.0.3. Like the PLG project, the source code of the implementation of the proposed extension is also publicly available[1]. This section will make several references to the classes contained in the implementation for reference.

In the implementation, certain simplifications have been made to PLG, in order to reduce its complexity. First of all, the model generator does not support or concern itself with data objects. Also, in order to be able to predict the number of branches in a generated gateway, all gateway patterns have been set to always generate two branches.

Figure 5.1 shows an overview of the classes used to implement the algorithm. Most associations do not have a direction or a multiplicity, except for the `Pattern`, the `PatternWeight` and `TargetWeight`. This was done to stress that it is not a one-to-one relationship, but rather that `DynamicRandomizationConfiguration` has as many `Pattern` instances as there are basic patterns being used, and that each of these has one weight attributed to it, which again has as many `TargetWeight` instances

[1] See `https://github.com/anders9310/plg`

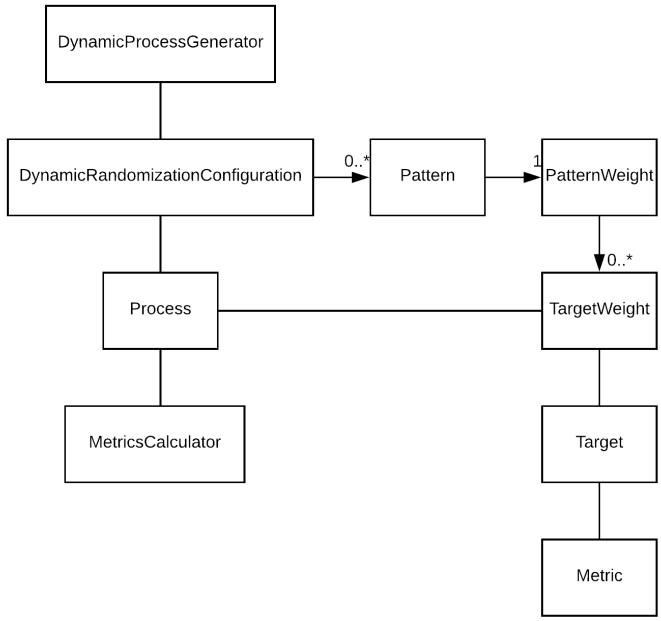


Figure 5.1: An overview of the main classes used in the implementation of the algorithm.

attached to it as there are metrics specified as obligations. The relationship between these classes is roughly equivalent to the relations shown in figure 4.2. In designing the architecture of figure 5.1, the principles of low coupling and high cohesion have been used as guidelines. This has lead to a design that has quite few associations per class. It should however be noted that the process is actually referenced in a few additional classes than what the diagram shows, and this is an implementation issue. However, a clean and uncluttered diagram showing where `Process` is mainly referenced was prioritized.

5.0.1 Implemented Distributions

The proposed generation algorithm, as presented in chapter 4 as algorithm 1, takes a set of obligations, which is a combination of a metric and its desired distribution, along with the particular properties of the distribution, such as its mean or variance.

For the implementation, a *Poisson distribution* has been chosen for all metrics. This has been chosen on the merits of its simplicity, as a particular Poisson distribution can be specified by its mean, and its variance will be the same. It also follows a somewhat even distribution resembling a normal distribution.

Since a Poisson distribution is a discrete probability distribution, it is only defined for values in $\mathbb{N}$. This is perfectly suited for metrics that also take only values in $\mathbb{N}$, such as the number of nodes or cycles. However, it does not work as well for metric that are defined outside of $\mathbb{N}$, e.g. $\mathbb{R}$. These metrics include most ones that are defined as ratios, such as the sequentiality or coefficient of connectivity. In order to also accommodate such metrics, a scaled Poisson distribution is used. More specifically, a Poisson distribution is scaled down by a factor 100 to fit it to metrics defined outside of $\mathbb{N}$. For example, for a distribution for the sequentiality of 0.46, a Poisson distribution of mean $\lambda = 46$ would be created and each interval divided by 100, so the new mean would become $\lambda_{scaled} = \frac{46}{100} = 0.46$. This also means that each metric defined in $\mathbb{R}$ can only be specified with a granularity of two decimals, such that specifying a mean of 0.123456 would only be interpreted as 0.12. Note that no assumptions are made on the validity of scaling down the Poisson distribution in this manner, but is purely a practical way of supporting such metrics.

5.0.2 Implementation of the `random()` Function

The `random()` function described in section 4.1 is implemented as the `poissonRandom()` function of the `Random` class. The implementation is based on inverse transform sampling, and as such runs in time proportional to the specified mean of the Poisson distribution. The implementation has been based on the inversion method presented in [34].

5.0.3 User interface

In addition to its user interface, PLG also offers a command line interface. However, none of these interfaces are supported by the implementation in its current state. Instead, the extension can only be run by accessing the interface of the `ProcessGenerator` directly, which is the same as shown in a Java fragment in [5]. However, the difference between running the traditional version of PLG and the proposed extension, is that instead of a `ProcessGenerator`, a `DynamicProcessGenerator` is used. This new type of process generator takes a different input than the tradition-alone, which reflects the input of the proposed extension. This can be seen in figure 5.2.

Since the proposed extension only supports Poisson distributions, which can be fully defined by specifying their mean, the `DynamicProcessGenerator` actually only takes pairs of metrics and the desired mean value of the distribution. When the algorithm initializes the generation of a process, these pairs are converted into targets, as they are described in chapter 4. This means that the implementation actually does not realize the notion of *obligations*, which define the actual input of algorithm 1.

Another important difference between algorithm 1 and the implementation is that there does not exist a part of the implementation that takes the number of process

```
for(int i = 0; i< numberOfModels; i++){
    Process p = new Process( name: "test" );

    Map<Metric, Double> inputs = new HashMap<>();
    inputs.put(Metric.NUM_NODES, 40.0);
    inputs.put(Metric.CONTROL_FLOW_COMPLEXITY, 10.0);

    DynamicProcessGenerator generator = new DynamicProcessGenerator(p, inputs);
    generator.randomizeProcess();
}
```

Figure 5.2: An Java fragment of the generation of several process models through the interface of the proposed extension. Notice the different type of the process generator from the one used in the Java fragment listed in [5]..

models to generate. Rather, this has to be done by generating a model through the `DynamicProcessGenerator` multiple times, as is done in figure 5.2.

5.0.4 Support for Metrics

An important aspect of the algorithm is the continuous calculation of the metrics of the model being generated, in order to make sure that the benefit gets it closer to the target values. As stated in section 4.3.5, any metrics that the generation algorithm supports as part of the input obligations must have its own metrics calculator. However, this is not the only reason why a metric might not be supported. As discussed in section 6.4, other factors also play a role. The tables 5.1 and 5.2 state which metrics are supported by the algorithm, and provide a short description of why they are not supported, if this is the case.

Most supported metrics have calculators that are trivial to implement, as they simply count elements in the model, or ratios between such counts. One metric that is more complex in this respect is the number of cycles, since cycles are not explicitly kept track of in the process model. In order to evaluate this metric, an implementation of Johnson's algorithm[25] from a third party is used. Information about the third party implementation is included in the publicly available implementation of this work.

If the reason for a metric not being implemented is that it is "not prioritized", this simply means that the implementation has been deemed overly complex compared to other metrics. This is often the case for control-flow related metrics, such as the diameter, which require an algorithm that can take this into account. This means that a third party algorithm would have to be found and adapted to fit the solution, or a custom one would have to be implemented, both of which has the risk of become significantly time consuming.

Several other metrics are not supported due to framework issues. This usually means that the designed algorithm is not suited for incrementally getting the model

closer to the target value by using the patterns of the framework from figure 3.4, In other words, the combination of the patterns and the proposed algorithm do not support the metric. This is however stated as a framework issue since, as proposed in section 6.4, these issues can probably be solved by changing the basic patterns. There are however situations where this is not enough, and the algorithm also has to improve, in which the algorithm design is stated as the issue.

5.0.5 Post-Evaluation of Metrics

In addition to the run-time calculation of metrics required to calculate the benefit of choosing patterns, the implementation also includes an integration to BPMeter[22], accessing its public API. This easily allows users to see the metrics of finished and exported models, and is what will be used to obtain the results of section 6.2. The number of metrics BPMeter supports is however limited, and only includes a subset of the metrics defined in [33].

5.0.6 Extensibility and Maintainability

Extensibility is the aspect of adding new features to the software without major changes to the design or implementation, and maintainability is the aspect of performing functional changes, and the ease of which this can be done. In this section, these two concepts will be used interchangeably to refer to the ease with which the algorithm can be extended or changed, for simplicity. This section lists some significant ways the algorithm is expected to be changed or extended, based on the issues discusses in chapter 6, and what it would take to implement such changes.

An important part of extending the capabilities of the algorithm is to support more metrics. If the algorithm can support the desired metric in principle, it can be added to the system by defining a value for it in the `Metric` enum, and implementing a method for calculating its metric in `MetricCalculator`. This means that the complexity of adding a new metric mostly depends on the implementation of its calculator.

An issue for supporting several metrics is the basic patterns used by PLG in figure 3.4. Section 6.4 suggests implementing basic patterns, or construction rules, equivalent of those used in [40]. This would firstly require a redesign of the rules to find the equivalent, and would also cause a couple of changes in the algorithm. First of all, the recursiveness of the generation framework would have to be changed to a more iterative one. More specifically, this means that the `ProcessGenerator` would no longer generate one single pattern recursively by calling inserting patterns for placeholders over and over, but would have to iteratively apply the construction rules to the model. Also, the framework would need a way of deciding where to apply each construction rule. What kind of scheme to use is more of a design issue than an implementation issue, but it should be possible to contain the implementation so that it is not dependent on any other parts of the system than `ProcessGenerator`.

Another suggested improvement is to redesign the algorithm for predicting the benefit of choosing patterns, to one that is capable of looking several steps ahead in the generation, i.e. is more clever. The implementation of such a design should be able to be contained in the `Benefit` class, or to only be dependent on this and the information contained there.

As stated in table 5.1 and 5.2, there is an implementation issue that keeps the density metric from being implemented. This is related to the rules when calculating the benefit, where a distinction is made between monotonically increasing metrics, and metrics that are not necessarily this, namely ratio-based metrics. This means that it is regarding any ratio-based metric (or rather any metric defined in $\mathbb{R}$) as non-monotonic. However, for the density metric, this is not the case for all but small model sizes, as it is monotonically decreasing for bigger ones. A solution to this is to create new rules when calculating the benefit that are general for all types of metrics.

5.0.7 Performance

The actual running time of the algorithm is discussed in section 6.3, in which it is shown that the running time is highly dependent on the size of the models generated. While improving this dependency, or changing the time complexity of the algorithm in general, requires a redesign of the algorithm, there are certain optimizations that can be readily made to reduce the running time.

The most obvious optimization, and the only one that will be described here, is reducing the time the algorithm spends calculating the metrics of the model being generated. By debugging the algorithm, it is possible to show that each metric is evaluated a whopping *170 times* for each new pattern that is generated. Ideally, the algorithm should only calculate each metric for the model once for every possible pattern, in order to predict the value of the metric if the pattern is generated, each time a new pattern is chosen. This would add up to ideally calculating the metric *5 times* each time a new pattern is chosen, which is 34 times less than the ideal number of times. The reason why this is not the case is mainly due to the fact that every time the value of a metric is read at any point during execution, it has to be calculated all over again. This means that every time it is logged, or even quickly checked by another class, it has to be recalculated. However, this can be improved by creating a caching solution, in which the `Process` caches the values of its own metrics, and invalidates this cache whenever a change to it happens, which is usually when a new pattern is added.

Metric	Is supported	Reason	Comment
Number of nodes	Yes		
Number of activities	Yes		
Number of gateways	Yes		
Number of AND-gateways	Yes		
Number of XOR-gateways	Yes		
Diameter	No (not prioritized)	Not prioritized	Can be supported, but was not prioritized due to implementation complexity.
Density	No	Implementation issue	Can be supported by the algorithm, but not by the current implementation of it. This is because the metric is monotonically decreasing when the number of nodes gets above a certain value, and is an implementation issue.
Coefficient of connectivity	Somewhat	Framework issue	The algorithm provides some support, but the metric can easily enter into a local maximum. This is a PLG framework issue, related to the patterns used.
Average degree of gateways	No	Framework issue	Can be supported, but due to simplifications to the generation framework, the only out-degree of a gateway possible is 2, which makes the average degree 1.5 for all finished models.
Maximum degree of gateways	No	Framework issue	Can be supported, but due to simplifications to the generation framework, the only out-degree of a gateway possible is 2.

Table 5.1: An overview of which metrics of table 3.2 are supported by the current implementation of the algorithm.

Metric	Is supported	Reason	Comment
Separability	No	Not prioritized	Can be supported, but was not prioritized due to implementation complexity.
Sequentiality	Somewhat	Framework and algorithm design issue	Only provides limited support since increasing or decreasing the metric often requires a basic pattern followed by an activity to increase or decrease, which means that the algorithm often can not anticipate how to change the value of the metric. This means that the algorithm will often enter into an endless loop, and is a problem related to a combination of the available patterns, and the cleverness of deciding the metric benefit of patterns.
Structuredness	No	Framework issue	Not supported as the framework patterns only support structured blocks, which means that this metric would only ever take a value of 1, since it is fully structured.
Depth	No	Not prioritized	Can be supported, but was not prioritized due to implementation complexity.
Connector mismatch	No	Framework issue	Since the framework only supports block-structured models, all connectors will always be matched, and the connector mismatch would therefore only ever take the value 0.
Connector heterogeneity	Yes		
Control flow complexity	Yes		
Number of cycles	Yes		
Token split	Yes		

Table 5.2: An overview of which metrics of table 3.3 are supported by the current implementation of the algorithm.

Metric	**Calculator time complexities**
Number of nodes	O(1)
Number of activities	O(1)
Number of gateways	O(1)
Number of AND-gateways	O(1)
Number of XOR-gateways	O(1)
Connector heterogeneity	O(N_G), where N_G is the number of gateways in the model
Control flow complexity	O(N_G), where N_G is the number of gateways in the model
Number of cycles	O($s^2 * \log s$), where s is the size of the model being evaluated. Johnson's algorithm [25] is being used.
Token split	O(N_G), where N_G is the number of gateways in the model

Table 5.3: The time complexities of the implemented metric calculators.

This page has been intentionally left blank

CHAPTER 6

Results and Discussion

The following sections discuss the different aspects of the algorithm's qualities and shortcomings. The following aspects will be in focus.

- Correctness
- Diversity
- Running time
- Support for metrics

6.1 Correctness

This section seeks to show the correctness of the algorithm, i.e. how well metrics of generated models hit their target values, as well as the distribution of metrics across the generated model collection. If assuming that a proper distribution of metrics is ensured if the algorithm hits the target values for each model, then the meaning of the correctness of the algorithm is *how accurately the algorithm hits the target values*. This means that if there exists a solution for which the generated model hits its target values, then the algorithm should generate such a solution.

The following subsections contains tests for different types of metrics, and are designed to show the merits and shortcomings of the algorithm when used on these metrics. Section 6.1.3 discusses the implications of the shortcomings seen in the tests, and proposes possible solutions for each of them.

6.1.1 Monotonic Metrics

The simplest metrics to handle for the algorithm should be monotonically increasing metrics, which usually take the form of *counts*, such as the number of activities or number of gateways. In order to determine the correctness of the algorithm for such relatively simple metrics, a test with three monotonic metrics is run. The input to the algorithm for this test can be seen in table 6.1.

None of metrics used as input for this test have any inherent dependency. They do, however, have some dependency in terms of the possible patterns the algorithm can produce. This is because 2 activities are always created when creating 2 AND- or

Table 6.1: Algorithm input with monotonic metrics.

Size of collection: 100
Obligations:

Metric	Mean value	Monotonic
Number of AND-gates	10	Yes
Number of XOR-gates	10	Yes
Number of activities	30	Yes

XOR-gateways through either a parallel execution, mutual exclusion or loop pattern. This means that the number of activities is always equal to, or higher, than the number of AND- and XOR-gates, such that *number of activities* $\leq$ *number of AND-gates* + *number of XOR-gates*. It is however always possible for the algorithm to reach a perfect solution for the number of activities, given that both the number of AND-gates and number of XOR-gates have hit their target value, and the number of activities has not yet, by creating a sequence and activity pattern a given number of times. In other words, there exists at least one solution where all three metrics hit their target, one of which the algorithm should choose.

Table 6.2: Results of the input in table 6.1.

Metric	Mean	Average ***absolute*** distance to target
Number of AND-gates	9.80	0.61 (6.2 %)
Number of XOR-gates	10.16	0.77 (7.6 %)
Number of activities	27.51	4.69 (17 %)

Looking at the results in table 6.2, it is arguable that the algorithm hits the target well for number of AND-gates and number of XOR-gates. The accuracy of the algorithm can be considered even better, when taking into account that the target value can take on odd values, while gateways are always generated in pairs of two, such that it is impossible to generate a model with an odd number of gateways.

While the results for the number of activities are not particularly bad, they are worse than for the other two metrics. The accuracy of the algorithm is low considering that it misses the target value by 17% on average. Noting that it most often hits a value lower than the target value, and considering that there is always a solution that makes the model reach or surpass the target value for the number of activities, these results are not optimal.

6.1.2 Monotonic and Non-monotonic Metrics

While most monotonically increasing metrics are fairly simple to handle, this is not necessarily the case for non-monotonic metrics. The challenge is that the the latter

can both increase and decrease, and may also contain local maxima on the way from the initial value to the target value. To test the correctness of the algorithm on such metrics in conjunction with monotonic ones, a test with the inputs of table 6.3 has been devised.

Table 6.3: Algorithm input for a mix of both monotonic and non-monotonic metrics.

Size of collection: 100
Obligations:

Metric	Mean value	Monotonic
Number of Nodes	40	Yes
Control Flow Complexity	10	Yes
Connector Heterogeneity	0.75	No

As can be seen from table 6.4, none of the metrics used hit the desired mean, the closest one being the connector heterogeneity, which produces a distribution with a mean value only 0.02 from the desired mean. Its average difference to the target value for each case is only 6.6%, which is only 1.6 percentage points above the tolerance of the generation algorithm (e of `reachedTarget()`, stated in section 4.1). From table 6.4, the variance of the distribution is significantly lower than the expected variance of a Poisson distribution, which is the same as the mean. This can also be seen from figure 6.2(c), which shows that the shape of the distribution is narrower than the reference Poisson distribution.

For the number of nodes and CFC, while the variance of the data is closer to its expected value for the connector heterogeneity, it is evident that the mean value is substantially off the mark. Compared to the goal of hitting the target value for each metric every time, an average difference of 13.2 and 20.1% is a large offset. It should also be noted that the control flow complexity usually hits a value higher than the target value, while the opposite is true for the number of nodes.

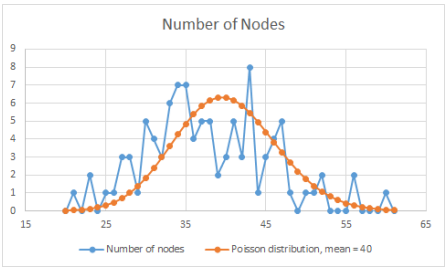
Table 6.4: Results of the input in table 6.3.

Metric	Mean	Variance	Average ***absolute*** distance to target
Number of nodes	37.78	58.49	5.27 (13.2 %)
Control Flow Complexity	11.99	9.11	2.01 (20.1 %)
Connector Heterogeneity	0.77	0.28	0.051 (6.6 %)

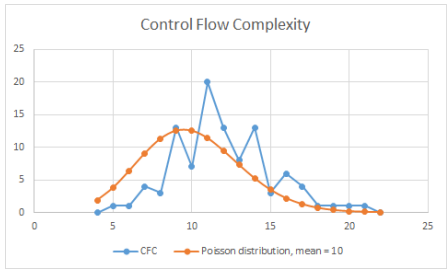
6.1.3 Discussion of Results

As can be seen from sections 6.1.1 and 6.1.2, while the correctness of the algorithm is fairly good, there is still room for improvements. This section analyzes the results to find the probable causes of its shortcomings, and proposes solutions to deal with them.

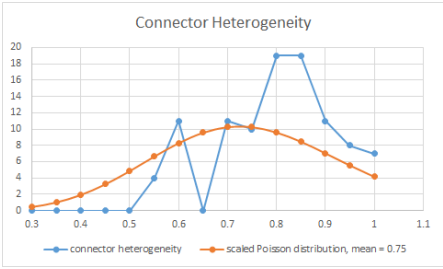
Figure 6.1: Distribution of the metrics in the collection generated from the input of table 6.3.



(a) Number of nodes



(b) Control Flow Complexity



(c) Connector Heterogeneity

6.1.3.1 Termination of Generation

In the test of section 6.1.1, the algorithm is shown to hit the target value pretty well for the two metrics regarding the number of gateways, but does not do as well for the number of activities, which on average hits a value 2.49 lower than the target value. Since the algorithm should be able to hit the target value for the number of activities by creating consecutive sequence and activity patterns, something is not working optimally in the algorithm. If, in the situation where the number of activities has not hit the target value yet, the algorithm never stopped, the algorithm would eventually reach the target value. This is because neither the sequence nor the activity pattern are disadvantageous for any of the metrics, so that if the algorithm never stopped, these would eventually be chosen. This indicates that the algorithm stops too soon. Since the termination of the generation of each model is controlled by the termination benefit part of the total benefit, this is likely where the problem arises.

Indeed there is a shortcoming in what controls the termination of the generation. Since this is generated as part of the benefit, the algorithm just gets a ”local” benefit of stopping from each of the metric target values, instead of a ”global” evaluation of whether it would actually be advantageous to stop the generation for the entire model. For the case of the test in section 6.1.1, what likely happens is that the two

metrics for the number of gateways hit their target value before number of activities. Since they have now reached their target value, their total benefits reflect that the algorithm should stop, while the benefits calculated for number of activities does not. However, since there are now two metrics that want the generation to stop, versus one that does not, the algorithm is going to favour patterns that brings the number of placeholders in the model toward 0, stopping the generation, even though there is still a metric for which the target value can be reached. In other words, the algorithm has an issue with stopping at the optimal time.

In order to stop at a more optimal point of the generation, a solution could be to look at the collective benefit of stopping the generation at every iteration. This can be done by looking at the average relative distance to all target values for all metrics from generating each pattern. If no patterns are contributing to a decrease in the average distance to all target values, the algorithm likely can not obtain a better solution, and should stop generation, by converting all placeholders in the currently generated model to skip patterns.

6.1.3.2 Local Maxima

When dealing with non-monotonic metrics, there is a risk of the algorithm being caught in a local maximum in terms of the benefit. This can happen if it is possible to generate a solution where the metric hits its target value, but no patterns show any benefit for the metric at the current point of generation. In such a situation, the algorithm can get stuck trying to reach the target value, but not know which pattern to choose to get there.

One example of such a situation is for the sequentiality metric, which is the ratio of the number of connections between non-gateway nodes to the total number of connections. In order to increase this ratio, the algorithm has to generate an activity pattern that can not connect to a gateway. While this is usually possible, it can happen that all remaining empty patterns in the generating models are between two gateways, such as "AND ; p_{empty} ; AND". Generating an activity pattern in place of the empty pattern would not increase the ratio, but decrease it. Instead, the algorithm has to generate more than two sequence patterns, and then activity patterns to fill their empty patterns. However, since the algorithm only looks one step ahead, it does not know this, and will easily be stuck in a loop of creating sequence and skip patterns, since neither of these decrease the value of the ratio. So the algorithm is stuck in a local maximum.

This shortcoming can be solved by introducing a smarter scheme for determining the benefit of choosing new patterns, which would have to be able to see several "moves" ahead. A possible solution that springs to mind when viewing generating a pattern as performing a move, is one borrowing inspiration from chess AIs, such as [7]. The commonalities between the generation algorithm here and such algorithms is that both have to look several moves ahead, and take a large number of different outcomes into account, which are determined in a stochastic manner. Other solutions are also possible, but proposing a specific solution is outside the scope of this thesis.

6.1.3.3 Impossible Metric Values

All metrics are only defined for a certain set of values. For example, the number of nodes in a model is restricted to the natural numbers $\mathbb{N}$. Likewise, the sequentiality can only take a value in $[0, 1]$. For the set of patterns used in PLG, the connector heterogeneity can only take values in $[0.5, 1]$. This can actually be seen in figure 6.2(c), where there are no occurrences of models with values outside this range, even though the scaled Poisson distribution with a mean of 0.75 that is used in the implementation is defined outside this range.

Fixing this shortcoming would not take a major redesign of the algorithm, but rather entails the algorithm having the defined range of all metrics, such that it knows not to set target values outside these ranges.

6.1.3.4 Empty Patterns

Another shortcoming of the algorithm is its use of empty patterns as placeholders, and their interpretation as empty strings. This usually manifests itself as a problem for certain metrics when calculating the benefit of generating patterns containing placeholders, since the patterns are necessary for increasing or decreasing their value, but generating such a pattern with empty strings instead of actual components does not affect the metric. One example is generating a sequence pattern for the diameter metric, which, among other patterns, is necessary for increasing the longest path of the model. However, when the new placeholder is interpreted as an empty pattern, the pattern does not actually add to the length of the path, since "component ; p_{empty} ; component" $\rightarrow$ "component ; component". This is also the case for metrics that rely on an actual component to be generated in each of the branches of gateway-patterns, before their effect can be seen. One such metric is the average degree of connectors, for the degree of each gateway-split will be 1 if there are only placeholders in each branch, and the number of branches if all placeholders were replaced by single activities.

One solution to this problem could be to introduce a more clever look-ahead during the calculation of the benefit, such that it looks several iterations ahead in the simulation, like proposed in section 6.1.3.2. This would allow the algorithm to predict the benefit for each metric, analogous to predicting a way out of the local maxima. Another solution would be to change the framework of PLG from generating placeholders altogether, and instead just generate activities, and treat these as placeholders instead. Such a scheme would start to look a lot like the one used in [40], where they define a set of *construction rules* that they apply to the model to transform it. In such a scheme, since no parts of the model would be undefined as the empty pattern at any time, the problem of placeholders is solved. It is however worth noting that if a metric still requires a certain series of generated patterns, a smarter look-ahead algorithm for determining the benefit is needed.

6.2 Diversity

In addition to having the metrics passed as obligations follow the specified distribution, it is also desirable to have a certain diversity in the metrics not passed as obligations. This means that the distributions of these other metrics should be somewhat spread out, following something that looks like a normal distribution.

In order to test the diversity aspect, the algorithm is run with just the number of nodes passed to it as an obligation. Instead of just choosing the same solution every time, the algorithm should ensure that each of the other metrics are also diverse. The input can also be seen in table 6.5, and the results are shown in figure 6.2.

Table 6.5: Algorithm input specifying the size of the generated collection.

Size of collection: 100
Obligations:

Metric	Mean value	Monotonic
Number of nodes	50	Yes

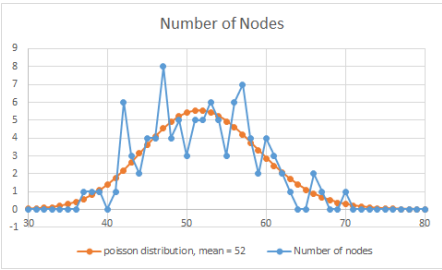
The diversity has been evaluated for the metrics in figure 6.2(a) through 6.2(e). For number of nodes in 6.2(a), the models follow a Poisson distribution as expected. When looking at the metrics of figures 6.2(b), 6.2(c) and 6.2(d), these also seem to follow a unimodal and Poisson-like distribution. The coefficient of connectivity in figure 6.2(e) is somewhat different in that it follows a distribution with a lower variance than the other metrics. By examining the analysis of real process models in [33], it becomes clear that this metric tends to follow a distribution with a low variance compared to the mean. This means that the narrow distribution of figure 6.2(e) can be considered satisfactory. Since the distributions of the metrics in figure 6.2 seem reasonably diverse, it can be concluded that the diversity of the metrics not passed as obligations is insured.

6.3 Running Time

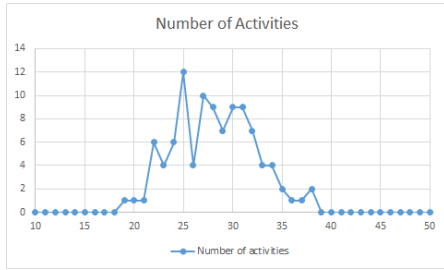
In section 4.4, the algorithm was argued to have a running time mainly dependent on the size of the produced model, as well as the time for calculating the metrics passed as obligations. In this section, this hypothesis is tried on a series of tests. The running time will also be evaluated for feasibility for use in practice. All tests are run on a computer with the specifications of table 6.6.

Table 6.6: Specifications of the computer used to run the tests.

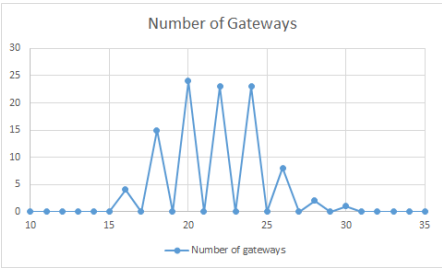
Model	Lenovo Thinkpad T440s
Processor	Intel(R) Core(TM) i7-4600U CPU @ 2.10 GHz 2.69 GHz
RAM	12 GB



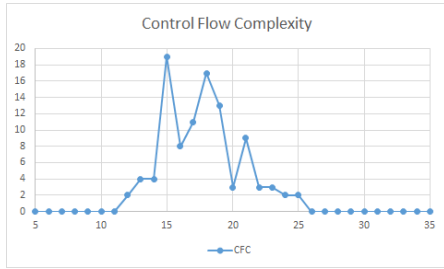
(a) Number of nodes



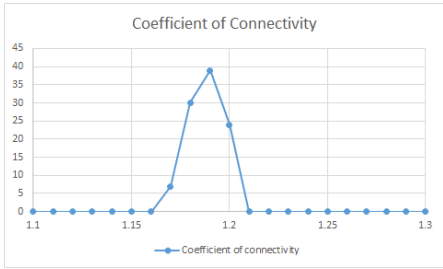
(b) Number of activities



(c) Number of gateways



(d) Control flow complexity



(e) Coefficient of connectivity

Figure 6.2: Distribution of metrics not included in the input of the test in table 6.5.

6.3.1 Size of Generated Models

According to the hypothesis, the algorithm should take longer when generating larger models. To test this, the algorithm is run three times - once when generating process models of 10 nodes, one for 32 (i.e. $10 * \sqrt{10}$) nodes, and another for 100 nodes. The test setup and results can be seen in table 6.7, which shows that there is indeed a big difference between the running times. This supports the hypothesis that the running time of the algorithm is greatly dependent on the size of the generated models, and in fact runs more than proportional to the size of the process model. In fact, it can

be shown that a polynomial regression line of the second degree can be very well fitted to the running time, as shown in figure 6.3. This is in line with the analysis performed in section 4.4, which states that the running time is proportional to s^2, i.e. not linear.

Table 6.7: Test results for different sizes of the generated models.

Generated models: 100

Number of nodes	Time elapsed per model
10	0.085 s
32	0.436 s
100	3.570 s

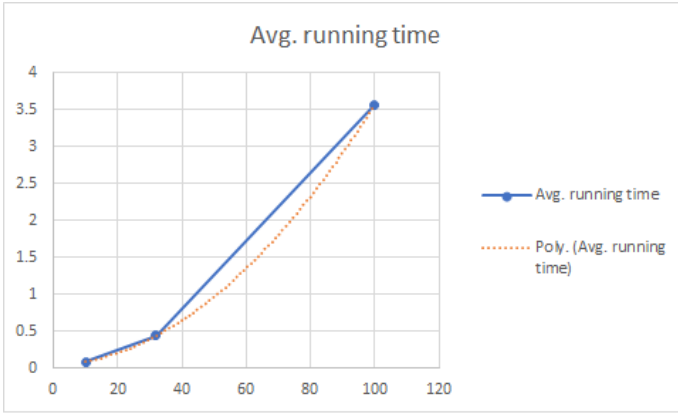


Figure 6.3: A polynomial regression line of degree 2 fitted to the running time of the algorithm in terms of the number of nodes generated.

6.3.2 Computation Time of Metrics

According to the analysis of section 4.4, the running time of the algorithm should also be dependent on the computation time of the metrics used. To test this, two metrics of different computational complexity have been chosen. The number of activities metric is very simple to compute, and takes only constant time in the implementation. The number of cycles metric is however a bit more complex to compute, and takes $O(s^2 \log s)$ time in the implementation, where s is the size of the generated process models. If the analysis is correct, it should take substantially longer to run the algorithm with the number of cycles than with the number of activities. In order to make the size of the process models irrelevant, both runs will be told to produce models of size 30 by also passing the number of nodes as an obligation.

The results of the test in table 6.8 clearly show a significant difference in the running times when using the two metrics, with the simplest one taking less time. The analysis is therefore correct in assuming that the metrics used in the algorithm are important for its running time.

Table 6.8: Test results for when using metrics with high and low computation time.

Generated models: 100

Metrics	**Time elapsed per model**
Number of nodes: 30 Number of activities: 20	0.547 s
Number of nodes: 30 Number of cycles: 5	0.919 s

6.3.3 Feasibility

In order to test whether the generation algorithm is feasible for use in practice running time wise, the running time of the algorithm is measured when running with several different metrics. In order to be feasible for use in practice, the algorithm must be able to construct a reasonable sized collection in an amount of time that can not disrupt the work of the researchers using it. For the purposes of this work, being able to generate a process model collection size of 2000 models in a reasonable time will be considered satisfactory for the feasibility. The number 2000 has been chosen since this is the combined size of the analyzed real-life process models in [33], which is the kind of process model sets that the generation algorithm is competing with for use in benchmarks.

The test setup and results can be seen in table 6.9. The test is run for 100 models, and the time to generate 2000 models is extrapolated from this (notice that this is the number of models, not the size of the models, so it is possible to assume linearity). This means that the algorithm will take 2.66 seconds $*$ 2000 = 5320 seconds, or 89 minutes, to generate 2000 models. If left to run overnight, or maybe even during lunchtime, this should not significantly impediment researchers' work.

It is however important to note that the models produced in this test are only of a modest size, and input producing larger models would likely have a significantly higher running time, as demonstrated in section 6.3.2. Extrapolating from the findings of that section, and assuming a squared dependence on the size, indicates that producing 2000 models of size 100 would take about nine hours, while producing 2000 models with 1000 nodes each would take about 170 hours, or 7 days. This means that while the algorithm is time-wise fit for use in practice for modestly sized models, it is possible that the running time of the algorithm exceeds what can be considered feasible when desired size of the models becomes too large. However, this test shows that the size of the models will have to exceed 100 models, and possibly be in the order of a thousand models for this to be the case.

Table 6.9: Results for the algorithm with several different types of metrics.

Generated models	100
Metrics	Number of nodes: 40 Control flow complexity: 10 Connector heterogeneity: 0.75
Time elapsed per model	2.66 s

As mentioned in section 5.0.7, it is however probably possible to reduce the running time of the algorithm significantly by implementing a caching mechanism for calculating the metrics, such that each metric will only be calculated 5 times for each new pattern that is chosen, compared to the 170 times it is currently calculated. While this should reduce the time significantly for all models, it still won't change the time complexity of the algorithm, such that generating very large models might still be unfeasible in many situations.

6.4 Support for Metrics

Section 6.1.3.3 discusses shortcoming of the algorithm in supporting metrics that are not defined for certain values. There are however also other shortcomings in the algorithm when it comes to supporting metrics, that are not as easily uncovered by tests. Compared to the metrics identified in [33], there are only a subset of those that are well supported by the algorithm.

Some metrics that can not be supported by the algorithm in its current form, are impeded by the generation framework and expressiveness of the CFC used by PLG. Since this only produces structured models, and never contains design errors such as the presence of deadlocks, and ensures liveness in the process, several metrics identified in [33] do not make sense to support, since they would usually only ever take on a single value. One example is the connector mismatch, which would only ever take a value of 0, since the CFG does not allow for mismatched gateways. Another example is the structuredness, which would always be 1, since the rules of PLG only allow for structured components, which are described in [33] when defining the metric.

In order to allow for support of such metrics, the CFG must allow for "incorrect" process models, which is to say models that contain design errors. One way of doing this is to introduce changes directly into the CFG. Another solution would be to make this a feature of a possible migration to a different framework of using *production rules* in stead of the CFG, as sketched out in section 6.1.3.4. By introducing an equivalent of the construction rules used in [40], the *soundness* property as defined in [38] would no longer be guaranteed. This means that the model can now contain deadlocks, and supports the generation of business process models that are transferable from any arbitrary Petri net.

Another shortcoming in terms of supporting metrics is the lack of support in

the implementation for handling different distributions for metrics. Different metrics often appear with different distributions in different real-life process model sets, which is evident from the real-life collections analyzed in [33]. One example is the coefficient of connectivity, which is shown to follow a significantly different distribution than other metrics in section 6.2. This means that in order to generate more realistic process model sets, it would be beneficial for the algorithm to be able to support a wider range of distributions.

It is also not possible for the algorithm to support monotonically decreasing metrics at its current state. This is due to an implementation issue where the algorithm is not told to stop if the metric falls below the target value, and is relatively easily fixable. It also has problems supporting metrics that are monotonically increasing, but not only take on values $m \in \mathbb{N}$, which is also due to an implementation issue. In order to be able to predict the number of branches in a gateway pattern, the number of branches has been set to 2 every time, which limits the range of values metrics based on the degree of nodes can take.

CHAPTER 7

Conclusion

This work proposes a generation algorithm that provides direct control of the characteristics of generated business process models, where the characteristics are specified as metrics along with their desired distributions. Each process model is generated with respect to a target value for each metric, which are produced from this distribution. The algorithm continuously calculates the probability of choosing among a set of basic production patterns, based on the benefit of choosing each pattern with regards to producing a model with the specified metrics. What patterns are beneficial or not is based on how much closer to the target values each pattern would get the model, and how the pattern contributes to the generation terminating at the right time.

The produced process model collections are generally in accordance with the specified characteristics. The results vary between metrics, and were on average within 6-20 % of the desired mean for different metrics, and had a good alignment with the reference distribution. The models also show a distribution resembling a normal distribution for characteristics that do not have a specified desired distribution, and shows an average running time indicating feasibility for practical use when generating models of around 100 nodes.

While the algorithm does achieve good results, there are still some serious limitations that makes impractical in its current state. A significant drawback is its limited support of metrics, mainly due to the basic generation patterns used. The algorithm also determines when to terminate generation non-optimally, and has problems handling local maxima and interdependence of metrics. However, despite its drawbacks, the algorithm is a nice first step towards developing a practically feasible method for generation of process models based on specification of the desired characteristics.

Future work

Chapter 6 discusses possible solutions to the aforementioned shortcomings, that will improve on the performance of the algorithm. These solutions to the most significant drawbacks can be summarized as introducing the following to the algorithm, as well as fixing the implementation issues identified in chapter 5.

- A new set of basic generation patterns that do not contain placeholder components, such that the model is non-ambiguously defined at all points of generation.

Patterns should also be as atomic as possible, so they can not be decomposed into other patterns.

- A smarter method for determining the benefit of choosing each pattern, by looking further ahead in the generation.
- A global notion of the benefit of terminating, where the generation does not terminate before an optimal solution has been found.

Future work on the proposed algorithm should focus on redesigning the algorithm to incorporate the above solutions. Likewise, subsequent work on similar algorithms should take note of the experiences made in this work, in search for a practically feasible method for generating process models with a high control of the output characteristics. Furthermore, the basic ideas of the proposed algorithm could also be applied in fields outside of business process modeling, such as graph synthesis, a field which seemed to lack a similar solution based on gradual optimization for generating graphs with certain characteristics.

APPENDIX A

Project Planning

When working on a project, having a plan for the progress towards reaching the end goal successfully is very important. By choosing not to work after some kind of plan, it is arguable that one is consciously planning to fail.

The project work on this thesis has been structured into two-week *sprints*, a concept that has been borrowed from the Scrum framework. Particularly for this project, it has meant that at the beginning of every sprint, a set of goals for the sprint has been set. Since some of the activities of the project work are not as tangible as delivering some functionality in a computer program, it has been the idea to make the results of the goals of each sprint as tangible as possible.

Before the beginning of each sprint, an informal sprint planning was also always performed. During this activity, the project plan as defined before the beginning of the previous sprint could be revised, to accommodate any necessary changes and to allow bigger tasks or more abstract goals to be broken down into smaller, more specific, tasks and goals. Right before the sprint planning, a meeting between the student and the adviser was scheduled. This served as a status meeting to keep the adviser up to date, and allowed for a discussion of what the next steps of the project should be, so this could be included in the revised plan. These meetings were the main contact between the adviser and student throughout the work period.

Working on the Thesis

The project work was mainly divided into a preliminary research phase, an algorithm design and implementation phase, and a writing phase. The preliminary research spanned the two first sprints, and the intention was that after the phase, a sufficient, if not complete, overview of the research field should have been obtained, such that the work of designing the algorithm could begin.

The algorithm design and implementation phase consisted of several sprints that each should deliver on the goals set for each sprint. While many projects have a somewhat set idea of what functionalities should be in the system, the main challenge of this system was not to implement what we wanted it to do, but rather to design *how* it should do it. Therefore, in contrast to a traditional system, each sprint was oriented towards both designing, implementing and evaluating a specific algorithm, which the subsequent sprint would improve on. The idea of this approach of iteratively designing and, hopefully, improving on the algorithm was to reduce the risk of no

functioning algorithm being created at the end of the phase. Instead, a simple but working algorithm was implemented during the first couple of sprints, and was then iteratively improved during subsequent sprints, but always working.

While working on the project, the design and implementation phase changed both in terms of the time allotted, and the goals, or milestones, set for it. While in the beginning, as seen from the initial plan in figure A.1, two milestones were set for the algorithm, where most of the different metrics should be supported after sprint 6, and that the capabilities should be expanded until the end of sprint 10, when the whole implementation should be done. As can be seen from the final plan in figure A.2, this layout changed quite a bit. First of all, the two aforementioned milestones were removed fairly early, and replaced by 5 iterations of the algorithm, where the fifth and final iteration was planned to finish after sprint 8, instead of ten. This also made the phases of the algorithm design and implementation more distinct from the writing phase, even though, as can be seen from the final plan, there was still some overlap in certain writing activities.

At the beginning of the design and implementation phase, the main idea was actually already very similar to the one in the final algorithm, namely to iteratively move toward a set of target values for each metric by recalculating the weights for each new generated pattern. The way of doing this was however initially a bit different from the approach taken in the end. Instead of calculating a predefined target value following a seemingly realistic Poisson distribution, the idea was that the algorithm would automatically terminate generation so that the generated models would follow such a distribution. This idea however turned out to be too difficult and complex, so after sprint 6, a different approach had to be taken, leading to the main idea of the final algorithm being adopted. Otherwise, the project work very well followed the gradual improvement of the algorithm for each completed sprint.

When the implementation of the algorithm was finished, the bulk of the thesis writing was undertaken, parallel to obtaining the final results. As can be seen from the final plan, several goals were not met in the sprints, but were still finished, thanks to some buffer time in the final sprint and at the cost of a Christmas vacation. While there were many differences between the initial and final project plan, this was the intention from the beginning. As more information about what would be the best way to process came to light, the project plan would change to accommodate this new information, by planning at the last responsible moment.

	Sprint number	Goals	Sprint description
Aug	1 preliminary research		Preliminary research in process properties and generation & dataset inspection
	2 preliminary research	**Preliminary research & project plan**	Analysis of preliminary properties in process models & project plan
Sep	3 Algorithm design and implementation		Analysis of process models & size parameter generation
	4 Algorithm design and implementation		Generation with CFC parameter
Oct	5 Algorithm design and implementation		Generation with size parameter
	6 Algorithm design and implementation	**Support for all generation parameters (metrics) & definition of these**	Generation with all parameters
Nov	7 Algorithm design and implementation & Writing	**thesis structure proposal**	Improve design & write
	8 Algorithm design and implementation & Writing		Improve design & write
Nov/Dec	9 Algorithm design and implementation & Writing		Improve design & write
Dec	10 Algorithm design and implementation & Writing	**Full implementation**	Improve design & write
Dec	11 Writing	**Thesis hand-in deadline**	Thesis writing

Figure A.1: The initial project plan. Each sprint has certain *goals*, which describe what the main purpose of the sprint is to achieve or produce, and a *sprint description*, which lists the smaller tasks that are part of the sprint, attached to it. The description is just meant for the student working on the project to read and maintain. Also, notice the few goals, which can be seen as milestones of the project at this early stage.

	Sprint number	Goals	Sprint description
Aug	1 preliminary research		Preliminary research in process properties and generation & dataset inspection
	2 preliminary research	**Preliminary research & project plan**	Analysis of preliminary properties in process models & project plan
Sep	3 Algorithm design and implementation		First iteration - algorithm implementation
	4 Algorithm design and implementation	**1st algorithm iteration**	First iteration - analysis & evaluation
Oct	5 Algorithm design and implementation	**2nd algorithm iteration**	base weight calculation & 0-generation bias problem & analysis efficiency improvement & algorithm evaluation & metrics section writing
	6 Algorithm design and implementation	**3rd algorithm iteration & Metrics section**	3rd iteration implementation and evaluation & metrics section writing
Nov	7 Algorithm design and implementation	**thesis structure proposal & 4th algorithm iteration**	4th iteration implementation & evaluation & Background writing & Thesis structure proposal
	8 Algorithm design and implementation	**Final algorithm iteration**	Final iteration implementation & evaluation & preliminary introduction section
Nov/Dec	9 Writing	**Generation algorithm section & related work**	Evaluation & generation algorithm + related work section
Dec	10 Writing	**Get final results & write results section & Generation algorithm section & Related work section & Design and implementation section**	
Dec	11 Writing	**Background section & Design and Implementation section & Related work & Final thesis**	Thesis writing

Figure A.2: The final project plan.

Bibliography

[1] Wil M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. 1st. Springer Publishing Company, Incorporated, 2011. ISBN: 3642193447, 9783642193446.

[2] Ruth Sara Aguilar-Savén. "Business process modelling: Review and framework". In: *International Journal of Production Economics* (2004). ISSN: 09255273. DOI: 10.1016/S0925-5273(03)00102-6.

[3] Leman Akoglu and Christos Faloutsos. "RTG: A recursive realistic graph generator using random typing". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2009. ISBN: 3642041795. DOI: 10.1007/978-3-642-04180-8{_}13.

[4] G BERTHELOT. "Transformations and Decompositions of Nets". eng. In: *Lecture Notes in Computer Science* 254 (1987), pages 359–376. ISSN: 16113349, 03029743.

[5] Andrea Burattin. "PLG2: Multiperspective process randomization with online and offline simulations". eng. In: *Ceur Workshop Proceedings* 1789 (2016), pages 1–6. ISSN: 16130073.

[6] Andrea Burattin and Alessandro Sperduti. "PLG: A framework for the generation of business process models and their execution logs". eng. In: *Lecture Notes in Business Information Processing* 66 (2011), pages 214–219. ISSN: 18651356, 18651348. DOI: 10.1007/978-3-642-20511-8_20.

[7] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. "Deep Blue". In: *Artificial Intelligence* 134 (2002), pages 57–83.

[8] Jorge Cardoso. "Approaches to Compute Workflow Complexity". In: *The Role of Business Processes in Service Oriented Architectures*. Volume 06291. July. 2006, pages 16–21. ISBN: 1862-4405. DOI: citeulike-article-id:13400941. URL: http://drops.dagstuhl.de/opus/volltexte/2006/821/.

[9] Jorge Cardoso. "How to Measure the Control-flow Complexity of Web Process and Workflows". In: *Workflow Handbook 2005* (2005), pages 199–212.

[10] P Chrzastowski-Wachtel et al. "A top-down Petri net-based approach for dynamic workflow modeling". eng. In: *Lecture Notes in Computer Science* 2678 (2003), pages 336–353. ISSN: 16113349, 03029743.

[11] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[12] Thomas Curran, Gerhard Keller, and Andrew Ladd. *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998. ISBN: 0-13-521147-6.

[13] Remco Dijkman, Marcello La Rosa, and Hajo A. Reijers. *Managing large collections of business process models - Current techniques and challenges*. 2012. DOI: 10.1016/j.compind.2011.12.003.

[14] Remco Dijkman et al. "Similarity of business process models: Metrics and evaluation". In: *Information Systems* 36.2 (2011), pages 498–516. ISSN: 03064379. DOI: 10.1016/j.is.2010.09.006.

[15] Dirk Fahland et al. "Instantaneous Soundness Checking of Industrial Business Process Models". eng. In: (2014). DOI: 10.1.1.418.2530.

[16] Emden Gansner et al. "Graphviz and Dynagraph Static and Dynamic Graph Drawing Tools". In: *In GRAPH DRAWING SOFTWARE*. Springer-Verlag, 2003, pages 127–148.

[17] Volker Gruhn and Ralf Laue. "Complexity Metrics for Business Process Models". In: *Proceedings of the 9th International Conference on Business Information Systems (BIS 2006)*. 2006, pages 1–12. ISBN: 1-4244-0475-4. DOI: 10.1109/COGINF.2006.365702.

[18] Kees van Hee, Natalia Sidorova, and Marc Voorhoeve. "Generalised Soundness of Workflow Nets Is Decidable". In: *Applications and Theory of Petri Nets 2004: 25th International Conference, ICATPN 2004, Bologna, Italy, June 21–25, 2004. Proceedings*. Edited by Jordi Cortadella and Wolfgang Reisig. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pages 197–215. ISBN: 978-3-540-27793-4. DOI: 10.1007/978-3-540-27793-4_12. URL: https://doi.org/10.1007/978-3-540-27793-4_12.

[19] Kees van Hee et al. "On the relationship between workflow models and document types". In: *Information Systems* (2009). ISSN: 03064379. DOI: 10.1016/j.is.2008.06.003.

[20] L. M. Hillah et al. "A primer on the Petri Net Markup Language and ISO/IEC 15909-2". In: *Petri Net Newsletter* 76 (2009). This article was also published at the Tenth Workshop and Tutorial on Practical Use of Coloured Petri Nets and CPN Tools (CPN '09), Århus, Denmark, Oct. 2009, pp. 101-120, pages 9–28. ISSN: 0391-1804.

[21] Karl Huppler. "The art of building a good benchmark". eng. In: *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5895 (2009). Edited by Nambiar et al., pages 18–30.

[22] Ana Ivanchikj, Vincenzo Ferme, and Cesare Pautasso. "BPMeter: Web service and application for static analysis of BPMN 2.0 collections". eng. In: *Ceur Workshop Proceedings* 1418 (2015), pages 30–34. ISSN: 16130073.

[23] Ingrid Jeacle and Chris Carter. "In TripAdvisor we trust: Rankings, calculative regimes and abstract systems". In: *Accounting, Organizations and Society* (2011). ISSN: 03613682. DOI: 10.1016/j.aos.2011.04.002.

[24] Tao Jin et al. "Efficient and Accurate Retrieval of Business Process Models through Indexing". In: *On the Move to Meaningful Internet Systems: OTM 2010: Confederated International Conferences: CoopIS, IS, DOA and ODBASE, Hersonissos, Crete, Greece, October 25-29, 2010, Proceedings, Part I*. Edited by Robert Meersman, Tharam Dillon, and Pilar Herrero. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 402–409. ISBN: 978-3-642-16934-2. DOI: 10.1007/978-3-642-16934-2_28. URL: https://doi.org/10.1007/978-3-642-16934-2_28.

[25] Donald B. Johnson. "Finding All the Elementary Circuits of a Directed Graph". In: *SIAM Journal on Computing* 4.1 (1975), pages 77–84. ISSN: 0097-5397. DOI: 10.1137/0204007. URL: http://epubs.siam.org/doi/10.1137/0204007.

[26] Oliver Kopp et al. "The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages". In: *ENTERPRISE MODELLING AND INFORMATION SYSTEMS ARCHITECTURE* 4.1 (2009), pages 3–13.

[27] M. Kunze and M. Weske. *Behavioural Models: From Modelling Finite Automata to Analysing Business Processes*. Springer International Publishing, 2016. ISBN: 9783319449609. URL: https://books.google.dk/books?id=HH4EDQAAQBAJ.

[28] Matthias Kunze, Matthias Weidlich, and Mathias Weske. "Behavioral similarity - A proper metric". eng. In: *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6896 (2011). Edited by Stefanie Rinderle-Ma, Farouk Toumani, and Karsten Wolf, pages 166–181. ISSN: 16113349, 03029743. DOI: 10.1007/978-3-642-23059-2_15.

[29] Michihiro Kuramochi and George Karypis. "Frequent subgraph discovery". eng. In: *Proceedings - Ieee International Conference on Data Mining, Icdm* (2001). Edited by N. Cercone, T. Y. Lin, and X. Wu, pages 313–320. ISSN: 23748486, 15504786. DOI: 10.1109/ICDM.2001.989534.

[30] Marcello La Rosa et al. "APROMORE: An advanced process model repository". In: *Expert Systems with Applications* (2011). ISSN: 09574174. DOI: 10.1016/j.eswa.2010.12.012.

[31] Kristian Bisgaard Lassen and Wil M P van der Aalst. "Complexity metrics for Workflow nets". In: *Information and Software Technology* 51.3 (2009), pages 610–626. ISSN: 09505849. DOI: 10.1016/j.infsof.2008.08.005.

[32] Andrew Lim, Wee-chong Oon, and Wenbin Zhu. “Abstract Towards Definitive Benchmarking of Algorithm Performance”. eng. In: (2008). DOI: 10.1.1.89.4216.

[33] Jan Mendling. *LNBIP 6 - Metrics for Process Models*. Springer-Verlag Berlin Heidelberg, 2008.

[34] B. J. T. Morgan and L. Devroye. “Non-Uniform Random Variate Generation”. und. In: *Biometrics* 44.3 (1988), page 919. ISSN: 15410420, 0006341x. DOI: 10.2307/2531615.

[35] “Redesigning Reengineering through Measurement-Driven Inference”. In: *MIS Quarterly* 22.4 (1998), pages 509–534. ISSN: 02767783. URL: http://www.jstor.org/stable/249553.

[36] Ichiro Suzuki and Tadao Murata. “A method for stepwise refinement and abstraction of Petri nets”. In: *Journal of Computer and System Sciences* (1983). ISSN: 10902724. DOI: 10.1016/0022-0000(83)90029-6.

[37] R. Tarjan. “Depth-first search and linear graph algorithms”. eng. In: *Siam Journal on Computing* 1.2 (1972), pages 146–60, 146–160. ISSN: 10957111, 00975397. DOI: 10.1137/0201010.

[38] W. M.P. Van Der Aalst et al. “Soundness of workflow nets: Classification, decidability, and analysis”. In: *Formal Aspects of Computing*. 2011. ISBN: 0934-5043. DOI: 10.1007/s00165-010-0161-4.

[39] Kees M. Van Hee and Zheng Liu. “Generating benchmarks by random stepwise refinement of Petri nets”. In: *CEUR Workshop Proceedings*. Volume 827. 2010, pages 403–417.

[40] Kees Van Hee et al. “Discovering characteristics of stochastic collections of process models”. eng. In: *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6896 (2011). Edited by Stefanie Rinderle-Ma, Farouk Toumani, and Karsten Wolf, pages 298–312. ISSN: 16113349, 03029743. DOI: 10.1007/978-3-642-23059-2_23.

[41] Zhiqiang Yan, Remco Dijkman, and Paul Grefen. “Business process model repositories - Framework and survey”. In: *Information and Software Technology* 54.4 (2012), pages 380–395. ISSN: 09505849. DOI: 10.1016/j.infsof.2011.11.005.

[42] Zhiqiang Yan, Remco Dijkman, and Paul Grefen. “Fast business process similarity search”. In: *Distributed and Parallel Databases* (2012). ISSN: 09268782. DOI: 10.1007/s10619-012-7089-z.

[43] Zhiqiang Yan, Remco Dijkman, and Paul Grefen. “Generating process model collections”. In: *Software and Systems Modeling* 16.4 (2017), pages 979–995. ISSN: 16191374. DOI: 10.1007/s10270-015-0497-6.

[44] Zhiqiang Yan, Remco Dijkman, and Paul Grefen. "Generating synthetic process model collections with properties of labeled real-life models". In: *Lecture Notes in Business Information Processing*. Edited by C. Ouyang and J. Y. Jung. Volume 181 LNBIP. Springer International Publishing Switzerland, 2014, pages 74–88. ISBN: 9783319082219. DOI: 10.1007/978-3-319-08222-6.

{2.7182818284}ϑφερτυθιοπσδφγηξκλ