A Structural Model Comparison for finding the Best Performing Models in a Collection

D.M.M. Schunselaar[1⋆], H.M.W. Verbeek[1⋆], H.A. Reijers[2,1⋆], and W.M.P. van der Aalst[1⋆]

[1] Eindhoven University of Technology,
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
{d.m.m.schunselaar, h.m.w.verbeek, h.a.reijers,
w.m.p.v.d.aalst}@tue.nl
[2] VU University Amsterdam,
De Boelelaan 1105, 1081 HV Amsterdam, The Netherlands
h.a.reijers@vu.nl

Abstract. An improvement or redesign of a process often starts by modifying the model supporting the process. Analysis techniques, like simulation, can be used to evaluate alternatives. However, even a small number of design choices may lead to an explosion of models that need to be explored to find the optimal models for said process. If the exploration depends on simulation, it often becomes infeasible to simulate every model. Therefore, for throughput time, we define a notion of *monotonicity* to reduce the number of models required to be simulated whilst the optimal models are still found.

1 Introduction

While improving or redesigning a business process, one can model each part of the process in various ways. Even if each part has a limited number of variants, the combination of options may cause an explosion of possible models. This set of possible models, we call a *model collection*. As a redesigner is often not interested in just any model, she would like to have qualitative and quantitative information on the models in the model collection such that she can select the most suitable models, assuming one or more relevant performance criteria. In this paper, we are particulary interested in the throughput time (sometimes called flow time, sojourn time, or lead time) of a model and the best models are those models having a significant lower throughput time than the other models. Unfortunately, brute-force approaches require the simulation of each model, which is a time-consuming endeavour. Therefore, we present a technique to reduce the amount of models needed to be simulated, whilst the best models are still found. We have chosen throughput time as this is a well-understood and often studied Key Performance Indicator (KPI) which can only be deduced from the dynamic behaviour of a model contrary to some other KPIs, e.g., number of control-tasks, which can be deduced from the structure of the model.

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

The aforementioned reduction is achieved by defining a *monotonicity* notion, which provides a partial order over the models. If we take the model collection on the left-hand side of Fig. 1, where each dot corresponds to a model, our particular monotonicity notion may allow for the partial order on the right-hand side of the same figure. Our monotonicity notion is based on the structure of the process models and gives a so-called *at-least-as-good* relation. If we can deduce monotonicity between a model M and a model M', then we know that the throughput time of M is at-least-as-good as the throughput time of M'. By having such a partial order, we can limit our search for the optimal models, i.e., to the models A and B in Fig. 1. We know that models not considered have poorer or equal throughput performance. In Fig. 1, if we have simulated A and B and A is better than B, then we know that all models connected to B via the partial order do not need to be simulated as A is *better than* all of them. In this paper, the structural comparison between two models is done using a *divide-and-conquer* approach. In this approach, each model is seen as a collection of weighted *runs* (runs are sometimes called process nets [1], or partial orders). Based on the structures of two runs, we can decide whether one run is at-least-as-good as the other run. Then, using the run weights, representing the execution likelihood, we can decide whether one model is at-least-as-good as another model.

This paper is organised as follows: In Sect. 2, we present our runs, models, and model collections. Our monotonicity notion is presented in Sect. 3. Finally, we conclude our paper with related work (Sect. 4) and the conclusions (Sect. 5). A more extensive version of this paper can be found in [2].

2 Model Collection

In this paper, a model collection consists of models, while a model itself consists of weighted runs [1] (see Fig. 2). A run specifies the partial order of activities needed to be executed for a particular case and does not contain any choices. A run consists of *vertices* which are labelled with activities. In Fig. 2, we have a run P_1 with a vertex v_1 labelled with activity a. Next to this, a run specifies the causal relationship between the vertices by means of edges. The edge between the vertices v_4 and v_5 in run P_1' means that e can only start once vertex v_4 executing d has been completed. We have abstracted from the transitive closure of the edges. The transitive closure of the edges we call paths. We have chosen this representation for our models as this fits better with

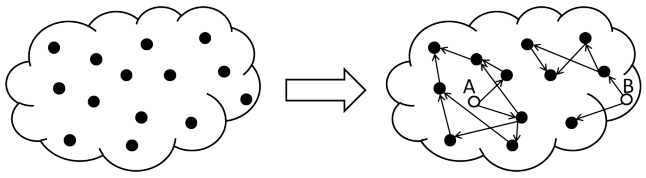

Fig. 1: Using monotonicity, we can transform the model collection on the left to the partially ordered model collection on the right. As a result, fewer alternatives need to be explored.

our divide-and-conquer technique. In [1], an algorithm is presented to transform a Petri net into a collection of runs. By limiting the number of times an iteration is executed, it is possible to obtain a finite set of runs.

A model consists of runs and these runs combined define the behaviour of a model. As not every run must be equally probable, we define a weight function (w) within the model. Furthermore, a model specifies which resources can execute which activity (ar), e.g., in model M in Fig. 2, activity a can be executed by resources r_1 and r_6.

Note that a resource can execute only one activity, but multiple resources can execute the same activity. We require that a resource can only perform a single activity to guarantee that we can compare the resource utilisation of both models and thus the queue time per activity based on the structure. This is due to the fact that even a small increase in the resource utilisation can have a significant effect on the throughput time.

A model collection consists of models, a set of activities (A), a set of resources (R), a random variable describing the inter-arrival time of new cases (K_a), and a function giving the processing time of each activity (K_{PT}). Note that our model collection is less general than most other model collections as our model collection is a body of scenarios for executing the same process.

3 Throughput Time

As mentioned before, we focus on the throughput time. The throughput time of a single case is the time between arrival of this case and the moment the case is finished. We consider the situation where the model and runs are in steady state.

To save space, within this section, MC is a model collection, M and M' are models from MC, P and P' are runs from M and M' respectively.

We are interested in the best models, that is, the models which have a significantly lower throughput time, which requires the comparison of two random variables. For this comparison, we use the Cumulative Distribution Function (CDF) of the throughput time which we also use as our notion of throughput time KPI. Note that our approach is not limited to this definition of the throughput time KPI. Any definition works as long as it is monotone, i.e., if the throughput time increases, then the KPI should decrease.

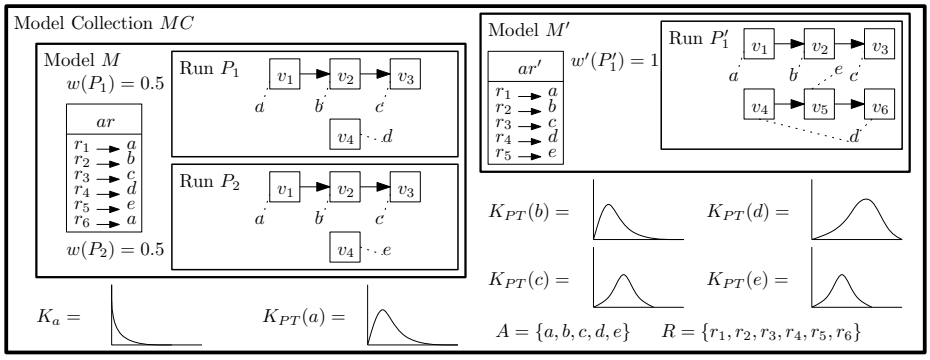

Fig. 2: An example model collection.

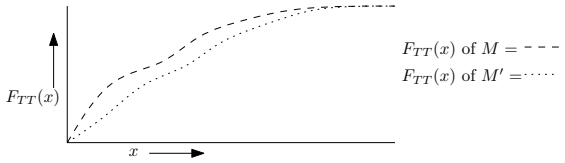

Fig. 3: Example throughput time KPIs for the models from Fig. 2.

Definition 1 (Throughput time KPI). *Let K_{TT} be the random variable describing the throughput time, let $\mathbb{P}(K_{TT} \leq x)$ be the probability that K_{TT} is at-most x, then the throughput time KPI, denoted by F_{TT}, is defined as: $\forall_x(F_{TT}(x) = \mathbb{P}(K_{TT} \leq x))$.*

Since our models consist of a collection of runs with a weight function, we define the throughput time of a model as the weighted sum of the throughput times of the runs. The at-least-as-good relation is defined as:

Definition 2 (At-least-as-good). *Let F_{TT} and F'_{TT} be two values for the throughput time, then the former is at-least-as-good as the latter if $\forall_x(F_{TT}(x) \geq F'_{TT}(x))$.*

By having that the throughput time KPI of M is above that of M', i.e., the probability that it stays below a certain point x is greater, we guarantee that in general M has a lower throughput time. It is, however, still possible that for an individual case the throughput time of M' is better than M. We are, however, interested in an overall comparison.

Taking the models from the collection in Fig. 2, Fig. 3 shows the possible throughput time KPIs for the models. By having that the KPI of M is above M', we conclude that M is at-least-as-good as M'. For the runs and vertices, we can draw similar graphs and also for runs and vertices it holds that if one KPI is above that of the other, then the former is at-least-as-good as the latter.

We define the at-least-as-good relation first between runs and show how this can be deduced based on the structures of the runs. Afterwards, we show how the at-least-as-good relation between runs can be leveraged to the model.

3.1 At-least-as-good runs

Informally when comparing the structures of the runs P and P', if P has fewer work, or has more flexibility in the order of executing activities, then P can do things faster than P' (assuming M has not less resources per activity). We operationalise this by comparing the vertices in P and P' (fewer vertices is fewer work), and by comparing the edges (flexibility in the ordering). Next to this, we need to make the requirement that we only compare vertices with the same label. When comparing two runs, we abstract from the respective models these runs are part of. In the comparison of two models, we shall elaborate on this.

Since which resource can execute which activity is defined on model level, we first introduce when a model M is at-least-as-good as M' with respect to the resource allocation, denoted by $M \geq_{ar} M'$. This is if every activity in M can be performed by at least the same resources as that activity in M'.

Definition 3 (Structurally at-least-as-good runs). *Let $M \geq_{ar} M'$, and let map be an injective mapping from vertices in P to vertices of P', then we say P is structurally at-least-as-good as P' given mapping map (denoted by $P \geq_s^{map} P'$) if and only if: (1) every vertex in P is mapped onto some vertex in P', (2) every path in P is mapped to some path in P', and (3) every vertex in P is mapped onto a vertex in P' labelled with the same activity. We say P is structurally at-least-as-good as P' (denoted by $P \geq_s P'$), if a mapping map exists such that $P \geq_s^{map} P'$.*

For Def. 3, we take two runs and compute the partial order, e.g., if we take P_1 and P_1' from Fig. 2, we obtain the partial orders in Fig. 4 (we have given the vertices from P_1' different names). Between these partial orders, we create a mapping which does not need to be unique, e.g., v_4 could also have been mapped onto v_6'. Taking also ar and ar' from Fig. 2 into account, we say P_1 is structurally at-least-as-good as P_1' since there exists a mapping such that: (a) each vertex in P_1 is mapped onto a vertex in P_1', (b) P_1 has fewer edges in the partial order, and (c) on model level, a can be executed by r_1 and r_6.

The throughput time of a vertex consists of the *queue time* and the *processing time*. Queue time is the time between the moment that a work item arrives at v and the moment the resource *starts* working it. The processing time is the time between when a resource *starts* working on a particular work item at v and the moment it is *finished*. This requires us to compare the queue times of the vertices. Therefore, we make the following assumptions:

1. the amount of arriving cases per time unit is exactly the same per run;
2. there is a single First In First Out (FIFO) queue per activity from which resources execute work items. This FIFO queue contains all the work items currently in the queues of the vertices labelled with a particular activity;
3. having more resources for an activity cannot increase the queue time of the FIFO queue for that activity if the amount of cases per time unit stays exactly the same.

Assumptions 1 and 3 allow us to compare the queue times for a particular activity. This is not yet enough for comparing the queue times between two vertices, e.g., it might be that the queue time on activity level is smaller, but, at a particular vertex, it could have increased. By having the FIFO queue, we prevent this from happening.

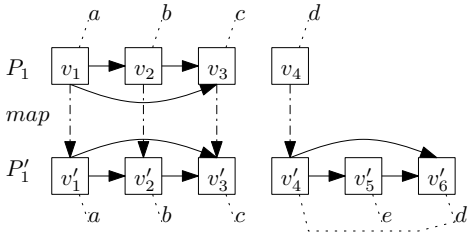

Fig. 4: The partial orders of the runs P and P' with a mapping map such that $P \geq_s^{map} P'$. Note that a mapping where v_4 is mapped onto v_6' would also have been fine.

3.2 At-least-as-good models

After showing the at-least-as-good relation between runs, we now define our at-least-as-good relation between models. The at-least-as-good relation between models holds if we can find a valid *matching graph* between the runs of the models. Graphically, a matching graph can be seen as a bipartite graph (Fig. 5). The runs of M are on the left-hand side and the runs of M' are on the right-hand side together with their weights. An edge between two runs indicates that the run on the left-hand side is at-least-as-good as the run on the right-hand side, e.g., P_1 is at-least-as-good as P'_1. As the weight of a run gives the probability of this run occurring it also gives the fraction of cases arriving for this run. Therefore, we have weights on the edges in the matching graph indicating the weight of the runs when they are compared. For instance, the 0.5 between P_2 and P'_1 indicates that in the comparison of P_2 and P'_1, we give them both a weight of 0.5.

As the weights on the edges in the matching graph indicate the weight of the runs when they are compared, we need to guarantee that the sum of the weights on the outgoing edges of a run is always that of the actual weight of that run. The same holds for the weights of the incoming edges of a run. The weights in the matching graph should be in $[0, 1]$. A (valid) matching graph is defined as:

Definition 4 ((Valid) Matching Graph). *Let $M \geq_{ar} M'$, then the matching graph, between M and M', denoted by $match_{M,M'}$, is a weighted collection of directed edges between vertices in M and vertices in M'. We say $match_{M,M'}$ is valid if and only if:*

- *if there is an edge between two runs in the matching graph, then the first is structurally at-least-as-good as the latter;*
- *the weights of the outgoing edges are the same as the weight of the run;*
- *the weights of the incoming edges are the same as the weight of the run.*

Using the matching graph, it becomes possible to structurally compare models with each other. If we are able to obtain a valid matching graph, we can conclude that one model is at-least-as-good as another model.

4 Related Work

By analysing redesign alternative encapsulated in a model collection, our work can be positioned on the intersection of *model collections*, and *performance evaluation*. We first discuss work from each of the two areas and then discuss work on the intersection.

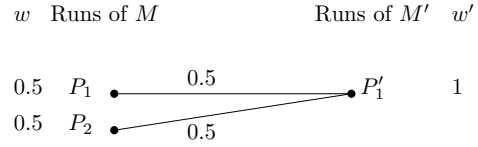

Fig. 5: An example valid matching graph.

In [3], the authors list the research areas within model collections. Often these model collections lack sufficient information for quantitative analysis, i.e., the context is missing, e.g., the arrival process of new cases, duration of activities, etc. If the model collection is viewed from a specific context, then our technique can be most beneficial in querying the collection of models. For instance, in PQL [4], the user can specify she is interested in models where an activity A is eventually followed by an activity B. As there might be a large amount of models returned from a query, our technique can be used to structurally order these models based on the throughput time. In this way, the user is immediately presented with the most promising models whilst adhering to the earlier specified structural requirements.

Within performance evaluation, the idea of monotonicity is not new and in queueing theory it has already been pursued [5]. In [5], the notion of monotonicity is similar but they focus on the parameters of the network and not the topology of the network. The work in [6] is similar to the work in [5] but now defined on continuous Petri nets. Since runs can be translated to Petri nets, this might be an interesting approach to use in the at-least-as-good relation between runs.

In [7], an approach is presented to evaluate when certain changes to the structure of the process model are appropriate. Starting from commonalities in reengineered processes, the paper deduces under which circumstances a change to the structure of the model is beneficial. The majority of the authors' ideas is not tailored towards throughput time but some ideas can be applied to our setting. These ideas are mainly on how resources perform their tasks.

So-called Knock-Out systems are discussed in [8] and heuristics for optimising these are defined. A Knock-Out system is a process model where after each task or group of tasks in case they are in parallel a decision is made to continue with the process or to terminate. The goal is to rearrange the tasks in such a way that the resource utilisation and flow time (throughput time) are optimised whilst adhering to constraints on the order of tasks encoded in precedence relations. By having an approach starting from a single model, this approach is not directly applicable to comparing two models.

In [9], a tool called KOPeR (Knowledgebased Organizational Process Redesign) is presented. KOPeR starts from a single model and identifies redesign possibilities. These redesign possibilities are simulated to obtain performance characteristics. This approach is not tailored towards directly comparing two models to determine which is at-least-as-good but our approach can be used to discard models prior to simulation.

In [10], process alternatives are analysed which have been obtained by applying redesign principles. Similar to the work in [9], our approach can aid in reducing the amount of to-be-analysed redesign options.

In our previous work, we have presented *Petra* a toolset for analysing a family of process models [11]. A family of process models is similar to a model collection but models are closer related. The work here can improve *Petra* by a-priori sorting the process models and only analyse the models most promising.

5 Conclusion

We have shown an approach to structurally compare the models within a model collection resulting in an at-least-as-good relation between models based on throughput time. This at-least-as-good relation can be used to minimise the effort to simulate a collection of highly similar models. This is particularly useful if redesigning an existing process where different improvement opportunities exist. Our approach poses a number of restrictions on the resources. In particular, we demand that resources can only execute a single activity and that they are truly dedicated to the process in question.

For future work, an interesting question is which of our assumptions can be relaxed to allow for the inclusion of a wider set of models to be considered. In particular, we want to look into whether runs have to be directed or whether they are also allowed to be undirected. This would allow us to compare different sequences of tasks and greatly increase our applicability. Furthermore, we want to extend the preliminary experimentation presented in [2] with models better reflecting reality.

References

1. Desel, J.: Validation of Process Models by Construction of Process Nets. In: BPM. Volume 1806 of Lecture Notes in Computer Science., Springer (2000) 110–128
2. Schunselaar, D.M.M., Verbeek, H.M.W., Aalst, W.M.P. van der, Reijers, H.A.: A Structural Model Comparison for finding the Best Performing Models in a Collection. Technical Report BPM Center Report BPM-15-05, BPMcenter.org (2015)
3. Dijkman, R.M., La Rosa, M., Reijers, H.A.: Managing Large Collections of Business Process Models - Current techniques and challenges. Computers in Industry **63**(2) (2012) 91–97
4. Hofstede, A.H.M. ter, Ouyang, C., La Rosa, M., Song, L., Wang, J., Polyvyanyy, A.: APQL: A Process-Model Query Language. In: AP-BPM 2013. Volume 159 of LNBIP., Springer (2013) 23–38
5. Suri, R.: A Concept of Monotonicity and Its Characterization for Closed Queueing Networks. Operations Research **33**(3) (1985) pp. 606–624
6. Mahulea, C., Recalde, L., Silva, M.: Basic Server Semantics and Performance Monotonicity of Continuous Petri Nets. Discrete Event Dynamic Systems **19**(2) (2009) 189–212
7. Buzacott, J.A.: Commonalities in Reengineered Business Processes: Models and Issues. Manage. Sci. **42**(5) (May 1996) 768–782
8. Aalst, W.M.P. van der: Re-engineering Knock-out Processes. Decision Support Systems **30**(4) (2001) 451–468
9. Nissen, M.E.: Redesigning Reengineering Through Measurement-Driven Inference. MIS Quarterly **22**(4) (1998) 509–534
10. Netjes, M.: Process Improvement: The Creation and Evaluation of Process. PhD thesis, Eindhoven University of Technology (2010)
11. Schunselaar, D.M.M., Verbeek, H.M.W., Aalst, W.M.P. van der, Reijers, H.A.: Petra: A Tool for Analysing a Process Family. In: PNSE'14. Number 1160 in CEUR Workshop Proceedings (2014) 269–288 http://ceur-ws.org/Vol-1160/.