J. Michael, J. Pfeiffer, A. Wortmann (Hrsg.): Modellierung 2022 Satellite Events,
Digital Library, Gesellschaft für Informatik e.V. 197

A Report on Automatic Generation of Petri Net Exercise and Exam Task Instances[1]

André Brandt, Marcellus Siegburg[2], Janis Voigtländer[3], Ke Wang

Abstract: We report on generators for different task types addressing Petri net concepts from a modeling lecture for undergraduate students. A focus is on how to control difficulty and intended insights about the subject matter on the learners' side. We explain the influence of provided configuration parameters for several task types on an exemplary instance each, and comment on presentation and implementation, as well as very briefly on exam experience.

Keywords: Petri nets; E-learning; Task generation

1 Introduction

To help teaching Petri net concepts to undergraduate students, and assess their understanding, we employ an e-learning setup with diverse types of exercise and exam tasks. We have implemented task types for different Petri net concepts (mathematical representation, concurrency, conflicts) and with different answer modi such as multiple-choice and matching tasks. Each task type is equipped with a generator that can produce a multitude of task instances. Each such generator is controlled by a set of custom configuration parameters. Such parameters can involve size constraints (e. g., how large the Petri net should be), structural constraints (e. g., whether certain graph patterns may or should appear in the Petri net), and more task specific constraints (e. g., whether certain forms of distractors should appear among the alternatives presented in a multiple-choice task). These parameters control the difficulty of generated task instances and allow to steer learning and understanding of specific aspects of the Petri net concepts under consideration. Setting the configuration values for a task generation run is performed by the lecturer. Each obtained task instance comes with a correct-by-construction answer that can be used in giving immediate automatic feedback on student submissions. But we have also used the setup with non-immediate grading as part of distant online exams, where the generation facilities were instrumental to providing individual task instances to students, basically eliminating potential for plagiarism.

In what follows, we discuss our Petri net task types, with a focus on their configuration parameters and how they can be used to adjust the level of challenge for students. We also

[1] Part of the work reported here was funded via:
Projekt „PITCH – Prüfungen innovieren, Transfer schaffen, Chancengerechtigkeit fördern“ (08/2021–07/2024),
Projektnummer FBM2020-EA-1190-00081, aus Mitteln der Stiftung Innovation in der Hochschullehre
[2] University of Duisburg-Essen, Faculty of Engineering, Germany, marcellus.siegburg@uni-due.de
[3] University of Duisburg-Essen, Faculty of Engineering, Germany, janis.voigtlaender@uni-due.de

briefly discuss our implementation strategy, and then conclude with some subjective as well as quantitative experience from exercises and an exam.

2 Task Type: Matching Representations of Petri Nets

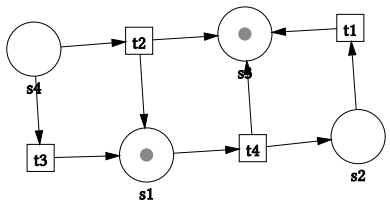
In the lecture, students are given a formal definition of Petri nets as a mathematical structure. This builds on set-theoretic constructions they have learned in a preceding discrete mathematics course. The role in our lecture is to provide a basis for formal definition of subsequent semantic concepts, but also to address frequent questions about the syntax of Petri nets, such as whether it is allowed to have an arrow directly from a place node to another place node, whether it is allowed to have a transition node without any outgoing arrows, etc. One task type used to practice the relationship between mathematical and diagrammatic representation is as follows: Students are shown a mathematical representation using the notation from the lecture, as in Fig. 1, and several Petri net diagrams, as in Fig. 3, and are asked which of the diagrams corresponds to the given mathematical rendering.

$N = (S, T, {}^\bullet(), ()^\bullet, m_0)$, where
$S = \{s_1, s_2, s_3, s_4\}$ and
$T = \{t_1, t_2, t_3, t_4\}$, as well as using the place ordering (s_1, s_2, s_3, s_4):
- ${}^\bullet t_1 = (0, 1, 0, 0)$
- ${}^\bullet t_2 = (0, 0, 0, 1)$
- ${}^\bullet t_3 = (0, 0, 0, 1)$
- ${}^\bullet t_4 = (1, 0, 0, 0)$
- $t_1^\bullet = (0, 0, 1, 0)$
- $t_2^\bullet = (1, 0, 1, 0)$
- $t_3^\bullet = (1, 0, 1, 0)$
- $t_4^\bullet = (0, 0, 1, 0)$

Moreover, $m_0 = (1, 0, 1, 0)$

Fig. 1: Mathematical representation of a Petri net

Let us discuss the main configuration parameters used for this task type, besides the obvious one controlling how many diagrams to present as possible choices. Several of them are also employed in other task types considered later on. Here we additionally mention with what concrete settings for the parameters the generator was called to obtain the task instance from Figs. 1 and 3, and the effect certain changes to those settings could have had.



parameter	value
`places`	`4`
`transitions`	`4`
`minTokensOverall`	`2`
`maxTokensOverall`	`2`
`maxTokensPerPlace`	`1`
`minFlowOverall`	`10`
`maxFlowOverall`	`10`
`maxFlowPerEdge`	`1`

Fig. 2: A Petri net abiding by the given configuration values

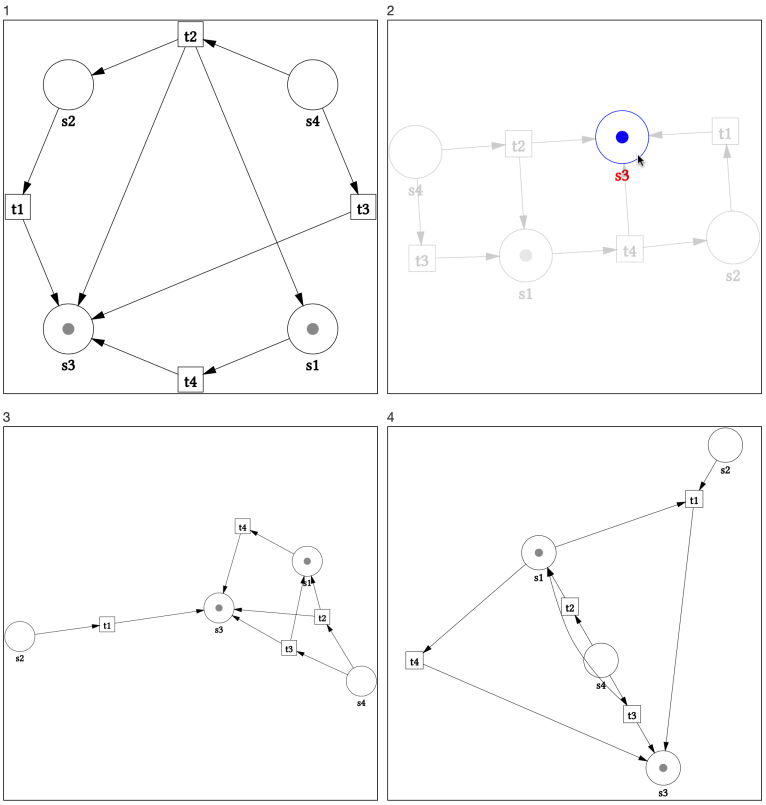


Fig. 3: Four Petri net diagrams, exactly one of which corresponds to Fig. 1

There are some self-explanatory size constraints which restrict the numbers of nodes and edges in a Petri net, as well as provide control over tokens and edge weights. These are illustrated in Fig. 2 by putting one generated Petri net alongside their specific values used. Note that we could have increased `maxTokensOverall` to `3` without changing `minTokensOverall`. This could have had two effects: First, some students might have received a task instance like the one from Figs. 1 and 3, with only two tokens in each net, while others would have seen three tokens per diagram. Second, even for a single task instance, it could in principle have been the case that some displayed diagrams contain two tokens, and others three tokens. The latter, however, would have been prevented here because we also set `tokenChangeOverall = 0` when generating Figs. 1 and 3. That makes sense in the current task type because redistributing tokens would simplify the generated task instances a lot: For example, since the initial marking is given as $m_0 = (1, 0, 1, 0)$ in Fig. 1, any diagram that does not have one token on each of s_1 and s_3 and no others, would immediately be disqualified from being the correct answer – without the student even having to look at the

pre- and postcondition weights in the mathematical representation.[4] In order to actually have some differences (apart from layout) between the presented diagrams, and to ensure that only one of them corresponds to the given mathematical representation, we use a setting `flowChangeOverall = 2` (and `maxFlowChangePerEdge = 1`).

One structural constraint setting used in the example is `isConnected = Just True`. Setting that parameter to `Just False` instead would have enforced that each of the four Petri nets offered as choices in Fig. 3 would have consisted of at least two disjointed graph components, while setting it to `Nothing` would have meant that we do not care. The same kind of "three-valued logic" applies to the configuration parameters `presenceOfSelfLoops`, `presenceOfSinkTransitions`, and `presenceOfSourceTransitions`, which were all set to `Just False` for the example. By allowing, or even enforcing, self loops and sink or source transitions to occur, we can already expose students to Petri nets containing these patterns, which will have a more interesting interplay with concepts like concurrency and conflicts later in the lecture. For example, a source transition can never be involved in a conflict.

Another parameter setting used here was `atLeastActive = 1`, basically preventing that any of the displayed Petri nets is deadlocked, despite that aspect not really having much of a conceptual impact on solving instances of this task type.

Concerning display, note that no weight numbers are being shown on the arrows in Figs. 2 and 3 because we set option `hideWeight1 = True`. This is the case throughout the paper to avoid issues with readability of overlapping labels. In our online setting, interactive highlighting features such as demonstrated in net 2 in Fig. 3 are available to students: marking nodes or edges in tandem with their corresponding labels when pointing on them.

Finally, we had `useDifferentGraphLayouts = True` and `graphLayout = [Sfdp, Circo, Neato, TwoPi]`, ensuring that each Petri net is displayed using a different GraphViz layout engine from a fixed set of choices. Setting `useDifferentGraphLayouts = False` and `graphLayout = [Circo]` would have meant that all four Petri nets in Fig. 3 are displayed in the current style of the first one, thus making it much simpler to work out – just visually – what the changes between them are, and thus which one corresponds to the mathematical representation in Fig. 1.

Another task type, similar to the one discussed above, and also implemented, works the other way around: displaying one diagram and several mathematical renderings, then again asking for correspondence.

3 Task Type: Finding Concurrent Transitions in Petri Nets

An important concept introduced in the lecture, for Petri nets but also with a more general modeling outlook, is concurrency. One task type used to practice detecting concurrency in

[4] In other task types, positive values for `tokenChangeOverall` make more sense, and are then accompanied by a positive setting for `maxTokenChangePerPlace`.

the context of Petri nets is as follows: Students are shown a Petri net, as in Fig. 4, told that it contains two concurrently activated transitions, and asked to identify the relevant pair.

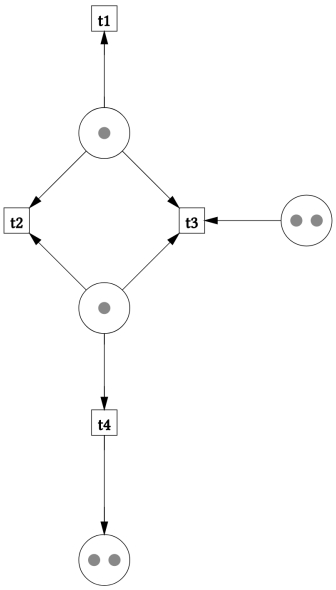


Fig. 4: A Petri net with two concurrently activated transitions

Several configuration parameters used for this task type are the same, and play the same role, as in Sect. 2. For example, there are again those mentioned in Fig. 2 as well as the `isConnected` parameter. The presence of a transition node without any outgoing arrows in Fig. 4 is not by happenstance but a consequence of setting `presenceOfSinkTransitions = Just True` in the configuration used when generating this example. Also, now `atLeastActive = 3` was used to avoid instances where only two transitions are activated at all, which would necessarily have made those the concurrently activated ones. That is, we wanted to have at least one further activated transition as a potential distractor. A new setting `hidePlaceNames = True` was used to simplify the display. In Sect. 2 that would not even have been an option because it would most likely make it impossible to always solve task instances (uniquely).

What might be called "change settings" (concretely `tokenChangeOverall = flowChangeOverall = 2` and `maxTokenChangePerPlace = maxFlowChangePerEdge = 1` in the configuration used when generating the example in Fig. 4) now play a slightly different role than before. Essentially, they control "how far away" the Petri net with concurrent transitions is from one that does not contain any concurrency. For example, we could thus deliberately generate only instances where the concurrency hinges on a single token (taking away that token would destroy the concurrency), or where it hinges on a single arrow's absence or weight (e. g., adding one arrow would destroy the concurrency).

A variant of the task type discussed above, also implemented, displays two Petri nets, one without concurrent transitions and one with a pair of concurrent transitions, and simply asks which net is which – without requiring students to pick out the specific pair of transitions.

4 Task Type: Finding and Explaining Conflicts in Petri Nets

Another important concept introduced in the lecture is the notion of two transitions in a Petri net being in conflict. One task type used to practice working with this concept is as follows: Students are shown a Petri net, as in Fig. 5, told that it contains two transitions in conflict, and asked to identify the relevant pair as well as the place(s) that is/are responsible for the

conflict. That is, they also have to identify all places that are joint preconditions for the conflicted transitions while not having enough tokens to fire both transitions concurrently.

There are again several configuration parameters that are already known from Sects. 2 and 3 – we do not repeat most of them here. To generate the example in Fig. 5, we again used `atLeastActive = 3`, but also `presenceOfSelfLoops = Just True`. Concerning the "change settings", similar comments as towards the end of Sect. 3 apply. A new parameter is `uniqueConflictPlace`. Setting it to `Just True` means that where the students have to identify all places bearing responsibility for the conflict, the correct answer will actually be a singleton. Distractors for the conflict and its origin can be configured using more advanced settings:

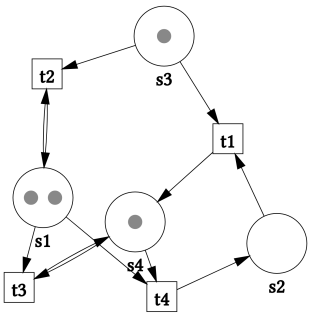


Fig. 5: A Petri net with two transitions in conflict (`t3` and `t4`)

Setting `addConflictCommonPreconditions = Just True` enforces that the two transitions in conflict have an additional place as common precondition, i. e., both having an incoming arrow from at least two same places. Such additional precondition places are meant to not be causes of the conflict. However, depending on the setting of `uniqueConflictPlace` from above, actual additional conflict-causing places could be enforced as well. Another setting that was used when generating the example in Fig. 5 is `withConflictDistractors = Just True`. It enforces the existence of at least one other pair of transitions, besides the conflicted pair, with non-disjoint preconditions. In the example, this resulted in the distractor pair `t1` and `t2` with common precondition `s3`. This pair has no additional common precondition places because `conflictDistractorAddExtraPreconditions = Just False` was set. Its common precondition place has only one token, and thus looks like a conflict, because `conflictDistractorOnlyConflictLike = True` was set.[5] What students would have to "discover" in the concrete instance from Fig. 5, in order to overcome the distractor pair, is that `t1` and `t2` are not actually conflicted, because `t1` is not even activated. But, of course, we as educators avoided having to handcraft the example to achieve this effect – instead relying on our generator and its declarative parameter settings.

In a slightly simpler version of the task type discussed above, the conflict-causing places do not have to be identified as part of the answer. Moreover, there is yet another task type, also implemented, which displays two Petri nets, one without a conflict and one with a pair of conflicted transitions, and simply asks which net is which.

[5] Setting `conflictDistractorOnlyConcurrentLike = True` instead would have resulted in more tokens on that place.

5 Implementation

While we discussed the task types on concrete examples in the preceding sections, each has an underlying generator. That is, for Sect. 2 we have a generator that from numeric and Boolean (or three-valued conditional) settings for `places`, `transitions`, etc., gives us many pairs of "Figs. 1 and 3", and likewise for Sects. 3 and 4. For example, Fig. 6 shows four random instances that were generated with the same configuration settings as Fig. 5. They are from a distant online exam, which is referred to in the next section, that employed literally hundreds of these (one per student). In preparation for the exam, similar exercise task instances had been provided to students, some generated with larger values for `places` and `transitions` and the token and flow numbers.

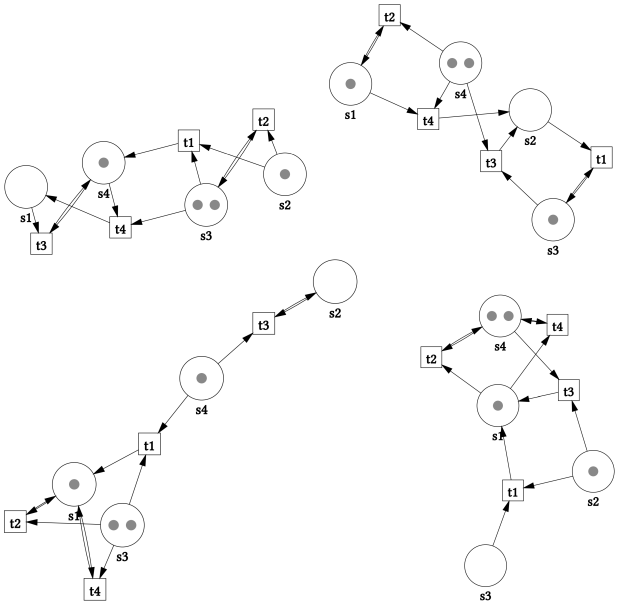


Fig. 6: Four of many different-but-alike instances used in an exam

It is probably apparent that randomness would not really do the job here. That is, a pure generate-and-test approach where Petri nets would be randomly generated (within given size constraints) and then tested whether they satisfy all the desires (like presence of certain distractors) and otherwise discarded, would not really be practical. Moreover, it would defy our correct-by-construction aspirations for task instances and feedback/sample solutions. Hence, we followed a more formal approach already adopted in previous work on generating task instances on the topic of UML class and object diagrams [KSV20, SV20]. It uses Alloy [Ja02, Ja11] to model the subject matter (back then, UML diagrams; now, Petri nets) along with various structural properties and semantic concepts on top. This gives an Alloy library, originally prototyped in [Wa19] and since then adapted and extended. Each task type can be thought of as a certain use case of that library. We let configuration parameters influence a logic formula built using predicates from the library, and sending that formula to the Alloy model checker and interpreting the returned outcomes gives us task instances. Initial experiments in this direction were the topic of [Br20], while the current work represents more fully developed task types and generators.

6 Conclusion

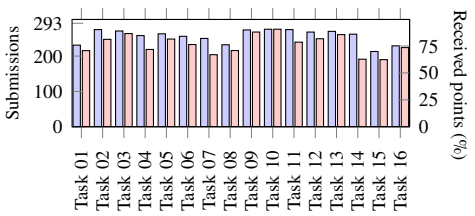


Fig. 7: Exam submissions made by students (blue) and point percentages received for those submissions on average (red).

Subjectively, judging from forum discussions with and among students, our generated exercise tasks on UML and Petri net concepts have been successful in furthering engagement with the material as well as pointing to areas of miscomprehension and need for additional practicing before the exam. We are also gaining more experience with using such generated tasks in actual exams. Fig. 7 shows some data from our latest installment (March 2022). Tasks 09 to 14 are Petri net task types reported on in this paper, while Tasks 02 to 08 are task types reported on earlier [KSV20, SV20], and the remaining three tasks are yet different. A deeper analysis is out of scope here, but even at a superficial glance we see interesting effects, such as the difference in results between Task 13 and Task 14, both "from Sect. 4", but only the latter requiring students to identify the conflict-causing place(s).

We also would like to systematically categorize our task types and configuration options by the learning objectives they address, also using appropriate taxonomies. A first foray in this direction was undertaken in [Sc20], but not for the specific task generators discussed here.

Bibliography

[Br20] Brandt, André: Automatische Generierung von Übungsaufgaben zu Petrinetz-Konzepten mittels Alloy. Bachelor thesis, University of Duisburg-Essen, 2020.

[Ja02] Jackson, Daniel: Alloy: A Lightweight Object Modelling Notation. ACM Transactions on Software Engineering and Methodology (TOSEM), 11(2):256–290, 2002.

[Ja11] Jackson, Daniel: Software Abstractions – Logic, Language, and Analysis, Revised edition. MIT Press, 2011.

[KSV20] Kafa, Violet; Siegburg, Marcellus; Voigtländer, Janis: Exercise Task Generation for UML Class/Object Diagrams, via Alloy Model Instance Finding. In: SACLA 2019, Proceedings. Springer, pp. 112–128, 2020.

[Sc20] Schweiger, Matthias: Analyse von Übungsaufgaben zu Petrinetz-Konzepten. Bachelor thesis, University of Duisburg-Essen, 2020.

[SV20] Siegburg, Marcellus; Voigtländer, Janis: Generating Diverse Exercise Tasks on UML Class and Object Diagrams, Using Formalisations in Alloy. In: MoHoL 2020. volume 2542 of CEUR Workshop Proceedings. CEUR-WS.org, pp. 89–100, 2020.

[Wa19] Wang, Ke: Exploring Petri net concepts through formalization in Alloy. Bachelor thesis, University of Duisburg-Essen, 2019.