Petri Net Project Report

Petrinet
Andrea Zatti and William Johansen
https://github.com/uiano/Petrinet

IKT445
13.12.2024

Abstract

This report documents the development and enhancement of the language for modeling Petri Nets created within the JetBrains MPS (Meta Programming System) tool. The project focused on improving and implementing new features in the language to facilitate the creation, visualization, and simulation of Petri Net models. Key advancements include the introduction of features such as inline comments for improved readability, compatibility with external tools for graphical visualization, and initial work to enable runtime simulation. Although the language successfully supports the fundamental concepts of Petri Nets, such as places, transitions, and token flows, future improvements are proposed to address limitations in handling complex configurations. Lastly, the project results are evaluated, the challenges encountered are discussed, and the next steps for further development are outlined, contributing to a solid foundation for continuing work in Petri Net modeling.

1 Introduction

This report presents the implementation and imporvments of a Petri Net modeling language using JetBrains MPS (Meta Programming System). The main task has been to improve upon the design and implementation of the domain-specific language that enables users to define Petri Nets. This report documents the process, covering the design decisions, technical implementation, and reflections on the project outcomes.

The structure of the report is organized as follows:

Language: This section details the language as implemented, including its purpose and features.

Solution: Describes in detail the implementation of additional features and improvements to the language. Each feature's "why" and "how" are described in detail and shown through examples.

Discussion: Evaluates the implementation, highlighting successful aspects, alternative design choices, and areas for improvement. This section includes empirical evidence of the language's correctness and discusses future steps.

Plan and Reflection: Reviews the project plan, issues, and lessons learned from the project.

2 Language

A *Petri Net* is a mathematical modeling language used to describe distributed systems. It is composed of places and transitions, and the links between places and transitions which are called arcs. Places represent conditions or resources, transitions represent events or actions, and arcs define the flow relationship between places and transitions.[2]

Tokens, which reside in places, symbolize the dynamic state of the system, and the execution of transitions moves tokens across places, modeling the system's behavior. Petri Nets are widely used for analyzing and simulating processes in fields such as workflow management, manufacturing, and communication protocols due to their ability to capture both concurrency and dependencies within systems.

Figure 1 is and example of a Petri Net with 4 places and 2 transitions.

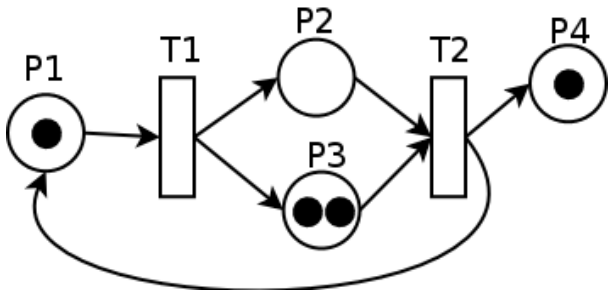


Figure 1: Petri Net with places **Px** and transitions **Ty**[1]

```
petrinet PN_1 {
  place P1 (1)
  place P2 (0)
  place P3 (2)
  place P4 (1)
  transition T1: P1 => P2, P3
  transition T2: P2, P3 => P1, P4
}
```

Figure 2: Example petrinet

Figure 2 represents the same Petri Net as figure 1, except it is modeled in using the MPS Petrinet language. The MPS language defines petri nets as a list of elements, places and transitions, contained in a Petrinet structure.

3 Solution

This section outlines the implementation of the Petri Net language, beginning with its core design and structure, followed by additional features that have been added to enhance its functionality and usability.

3.1 Structure

The structure of the Petrinet language is designed around the root concept `Petrinet`, which serves as the top-level container for all elements in a Petrinet model. The `Petrinet` concept contains a list of child elements, each of which can either be a `Place` or a `Transition`. Figure 3

```
concept PetriNet extends    BaseConcept
                implements PNIdentifier
                           ScopeProvider
                           IMainClass

  instance can be root: true
  alias: <no alias>
  short description: <no short description>

  properties:
  << ... >>

  children:
  elements : PetrinetElement[0..n]
```

Figure 3: Petrinet root concept

A `Place` contains the value of tokens associated with the said place and a `Transition`

contains references to all input places and output places. This simple hierarchical structure enables the creation of any Petrinet.

3.2 Syntax

Since the structure is defined as a list of elements, the syntax is also derived from this. Each non-empty line within the editor is a `petrinetelement`. The editor for the petrinet is shown in Figure 4. Figure 2 gives an example of a petrinet model, here you can clearly see how each line describes one element.

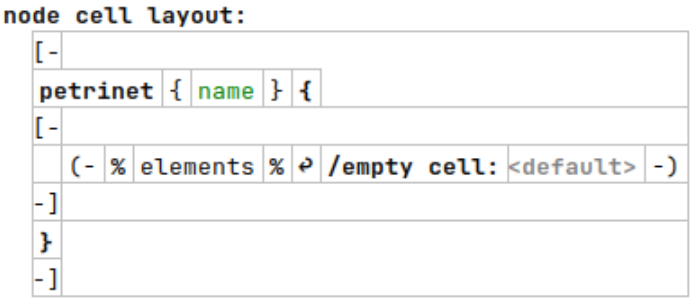


Figure 4: Editor for concept *Petrinet*

The property `elements` of the concept `Petrinet` is placed in a collection with each non-empty new line being a new child of the collection.

Each element may either be a `Transition` or a `Place`, both of which have different syntax. The syntax for `Transition` is shown in Figure 5 and the syntax for `Place` is shown in Figure 6.

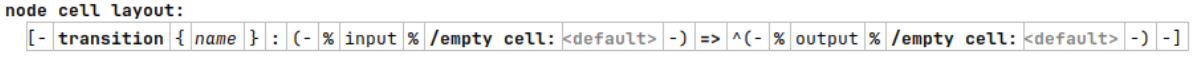


Figure 5: Editor for concept *Transition*

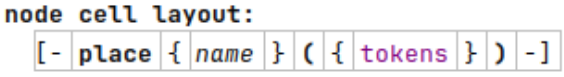


Figure 6: Editor for concept *Place*

3.3 Line comments

A new language feature for line comments has been introduced to help increase the readability of models. Line comments allow users to annotate their Petri Net designs directly within the editor, providing a way to explain, document, or clarify specific parts of the model. This feature is particularly useful for collaborative projects or when revisiting models after some time, ensuring that the intent and reasoning behind design choices are easily understood.

A comment is written using the symbol `#`. The string following the comment symbol does not enforce any constraints.

The `CommentLine` concept inherits from the same `PetrinetElement` as `Place` and `Transition`, meaning it inherits the identifier `name`, which has constraints associated with it. So to *disable* this constraint for the comment string, a new constant true constraint was written as an aspect for the `CommentLine` concept that would overwrite the inherited constraint, effectively *disabling* it.

3.4 Tests

At the start of the project, the existing tests for the Petri Net language were not functioning. This issue was most-likely caused by the fact that the test where written using a different version of MPS than we did. This difference might have introduced changes that disrupted compatibility with the test framework. This was resolved by refactoring some test and some minor configuration changes.

Once the initial tests were fixed, new tests were implemented to ensure the correctness and reliability of the new features added during the project. In particular, tests were created to validate the functionality of line comments, ensuring they behaved as intended.

```
Editor test case typeCommentLine
description: no description
before: petrinet abc {
  place a (1)
  <cell >
}
result: petrinet abc {
  place a (1)
  # b
  place c (2)
}
code:
  type "#b"
  invoke action -> Insert
  type "place c(2)"
```

Figure 7: Editor test for line comment

Having many good and wide covering test is very important to provided a solid foundation for further development and maintenance.

3.5 External tool

We have also implemented compatibility with an external online tool for the visualization of Petri nets. The tool in question is the following: `http://apo.adrian-jagusch.de/`. This tool was specifically chosen for its graphical simplicity, while at the same time being

effective and useful in illustrating fundamental concepts in this field, such as cycles or "well" states from which it is not possible to exit. To integrate this tool, we extended our text generation system to support the creation of files in the compatible `.apt` format. This allows us to load our Petri net into the previously mentioned tool, facilitating its visualization and analysis.

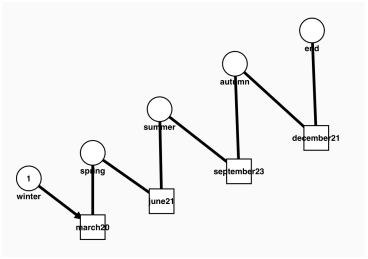
The `.apt` file is a special extension created by the tool's developer. All instructions on how to generate the text and create the file have been included in the updated section of the manual. Therefore, it does not make sense to repeat them in this section. In the following images, the entire process of creating and generating the graphical model of our Petri net can be observed.

```
petrinet seasons {
  place winter (1)
  place spring (0)
  place summer (0)
  place autumn (0)
  place end (0)
  transition march20: winter => spring
  transition june21: spring => summer
  transition september23: summer => autumn
  transition december21: autumn => end
}
```

(a) First code in PetriNet

```
.name "seasons"
.type PN

.places
winter
spring
summer
autumn
end

.transitions
march20
june21
september23
december21

.flows
march20: {1*winter} -> {1*spring}
june21: {0*spring} -> {1*summer}
september23: {0*summer} -> {1*autumn}
december21: {0*autumn} -> {1*end}

.initial_marking {1*winter}
```

(b) Generated text file



(c) External tool result

3.6 Documentation

The README and the manual had a structure and content that were unclear and insufficiently detailed. Therefore, we decided to enhance them with more information and explanations to provide new students with a clearer and more solid foundation to start from. This approach helps reduce the time required to understand the initial information about the project. Both the manual and the README are available for consultation on the project's GitHub page. Furthermore, the previous `todolist` file has been replaced with GitHub's Issues feature, where all the previously listed `todolist` items have been transferred, along with the addition of some new ones.

3.7 Runtime

An attempt was made to create a runtime for the Petri Net language, enabling the execution and simulation of Petri Net models directly from the MPS environment. The runtime aimed to interpret the structure and semantics of the language, simulating the token flow between places based on the transitions and their conditions.

The implementation consists of creating a runtime instance for a selected petrinet. Runtime

concepts for places and the petrinet itself was created to maintain any changes occurring during runtime, transitions did not have a runtime counterpart since they are fully static.

The runtime behavior was run via the MPS plugin feature, which enabled the user to simply right-click and "Run Petrinet" at any selected petrinet.

The implementation was mostly successful, with most of the runtime logic functioning as intended. However, there are currently issues with initializing the runtime instance of the Petri Net. Specifically, these problems arise during the setup phase, where the Petri Net's structure and initial token configuration need to be translated from the model into a runtime instance.

These initialization issues prevent the runtime from running, limiting its practical usability at this stage. Addressing these challenges will require further debugging and potentially reworking parts of the initialization process to ensure compatibility with the language model. Despite these setbacks, the progress made in the runtime implementation provides a promising foundation for future development.

4 Discussion

4.1 Coverage

The implemented language successfully covers most aspects of Petri Nets, providing the fundamental elements necessary to model and simulate these systems. Core concepts such as places, transitions, and tokens are well-represented, and the language supports the modeling of token flow through the system based on transitions and their input and output places. This allows users to create Petri Nets that capture concurrency, dependencies, and system dynamics effectively.

However, one of the most significant limitations is the lack of a more detailed mechanism to describe the relationships between transitions and places, especially in cases where there are multiple inputs and outputs. The current implementation relies on basic references, which work well for simpler configurations but may not fully capture the complexity of systems with intricate conditions or priorities governing the flow of tokens.

Future enhancements could address this limitation by introducing a way to specify the weight of inputs/outputs on a transition or other constraints to better define the behavior of transitions with multiple input and output connections. These additions would make the language more expressive and bring it closer to fully covering the theoretical capabilities of Petri Nets.

4.2 Positive aspects

The Petri Net language implementation has several positive aspects that contribute to its utility and appeal.

Simplicity: The language is designed with a straightforward structure, using a root Petrinet concept containing Place and Transition elements. This simplicity ensures that

users can quickly understand and start working with the language without requiring extensive training or background knowledge.

Readability: Models created with the language are highly readable, even for those unfamiliar with the specific implementation. Clear representations of places, transitions, and their connections make it easy to interpret the structure and behavior of a Petri Net at a glance. The addition of features like line comments further enhances readability by allowing annotations to clarify design decisions or document specific parts of the model.

Ease of Use: The language's editor and features are intuitive, making it easy to create, modify, and manage Petri Net models. Elements are simple to define and edit.

4.3 Negative aspects

The project has been active for many years and features a well-established structure. However, the only negative aspects we found at the moment are more details than actual issues. Nonetheless, addressing them could significantly enhance usability for the end user.

Specifically, the first issue we identified concerns the scalability of the language. If one wanted to create Petri nets with a large number of states and transitions, writing the code would become very cumbersome, as there is currently no way to generate, for instance, many similar elements using a more concise code. Another aspect that could greatly simplify the user experience is the integration of a graphical tool. At present, external tools are relied upon, but having a more intuitive graphical interface instead of writing code would broaden the project's potential audience considerably. Finally, some basic features of Petri nets are still missing. For example, it is not possible to specify that a transition consumes a number of tokens other than 1, making it impossible to model processes that alter the quantity of tokens during execution.

4.4 Future steps

The most relevant future developments primarily involve completing the implementation of the runtime and ensuring its proper functionality. Unfortunately, we were only able to implement it partially and did not have time to make it fully operational. However, we believe that the work required by future students will not be particularly demanding, as it will mainly involve identifying and fixing the key error we missed. As for the parser, in addition to addressing the issues related to its compilation, which is currently not functioning correctly, it would be beneficial to add more detailed and clear error messages. This would greatly facilitate debugging and make the parser more robust and intuitive for those who will use or work on it in the future.

5 Plan and reflection

5.1 Plan and time usage

We set the goal to have at least one meeting every week, usually Friday afternoon after the seminar. We managed to maintain this schedule quite well, only missing a few meetings due to other priorities getting in the way. During these meetings we would usually work through assignments or continue working on the project.

As we quickly discovered, most task in MPS takes more time that one would first assume. What would to us seem like trivial tasks could actually be quite difficult and thus take much longer time than expected. This paired with out inexperience with the quite complex tool that is MPS made it so that any changes took longer than expected.

5.2 Project management

To manage the task for the project we used GitHub Issues to describe and delegate tasks. We would often go though the issues list and update it as we discovered more issues or fixed issues, making it a constantly updating todo-list.

A lesson learned could be that it is usually better to overestimate than underestimate.

6 MPS learning quiz

1: How make can you make sure new features does not brake old features?

Write good tests and run then after any changes.

2: How to navigate through the key elements and structure of a project in MPS?

The quickest way to navigate between elements is by using the command to go to a symbol's declaration. Place the cursor on the desired symbol and press `Ctrl+B` to jump directly to its declaration.

References

[1] Wikipedia contributors. *Petri net*. Aug. 2024. URL: https://en.wikipedia.org/wiki/Petri_net#Formulation_in_terms_of_vectors_and_matrices.

[2] Carl Petri and Wolfgang Reisig. "Petri net". In: (2008). DOI: 10.4249/scholarpedia.6477. URL: http://www.scholarpedia.org/article/Petri_net.