PetriNet - Project Report

Group - PetriNet
Pedro Alves, Lukasz F. Mrozik, Katrina Lie
https://github.com/uiano/Petrinet.git

IKT445
16.12.2021

Abstract

The projected started with an implemented version of the language PetriNet in MPS. We implemented new constraints and new intentions as well as a interpreter. We also added new examples coded in PetriNet, as well as a cleanup of the documentation and general improvement of the manual.

1 Introduction

This project look at the PetriNet language and it is written in JetBrains MPS (Meta Programming System). The language has already been implemented, but is missing some elements, and that is what we will look into in this project. A todo list is given with multiple tasks that should be completed in order to finalize the creation of the language.

The todo list consisted of these elements:

- improve manual
- MOF diagram for structure
- distribute some samples with the plugin (and refer in manual)
- find a real external PetriNet tool for the text generation (and put into manual)
- sort between manual and README
- Could try to run build after generating CUP/FLEX
- build debugger for Petri net runtime (debug language)
  - Debugger = abstract exec engine
  - debugger.steps: single, into, out, watch (value)
  - stack frames? define by structure of statements
  - watches: hide system-level variables?
- add possibility to comment lines
- handle that the same place could appear twice in transition in or out (in execution)
- test references outside the petri net
- find better way for project path than absolute path
- remove all warnings and notifications
- create unit tests for the parser
- make TODO items into issues

We chose tasks from the list to work on depending on if it seemed to be within our understanding of MPS and PetriNet, and how much time we had to choose more challenging tasks.

First, a description of the language will be provided before looking into how the language is created and works by explaining the structure, constraints, syntax and semantics. Then the implementation that we applied is discussed. Finally we reflect on the project and the plan we had.

2 Language

The language PetriNet, or place/transition (PT) nets is a language that is based on mathematical models. PetriNets it self is a state-transition system. It consist of places and transitions, and has arcs and tokens that decides the flow of the network [1]. The language is specified with structure, an editor and transformation, which will be explained further in the solution 3.

Places can hold x-amount of tokens, the transition makes the tokens go from place to place, depending on where the arc leads. Looking at figure 1 the places are represented as the blue circles and the token is the black dot inside "Spring". Transitions are marked as "T" and the arcs are the arrows between the transitions and places. For a transition to be activated all places in the input of an transition must hold at least one token. When a transition is activated the token is fired from one place to the place connected to the output of the transition.

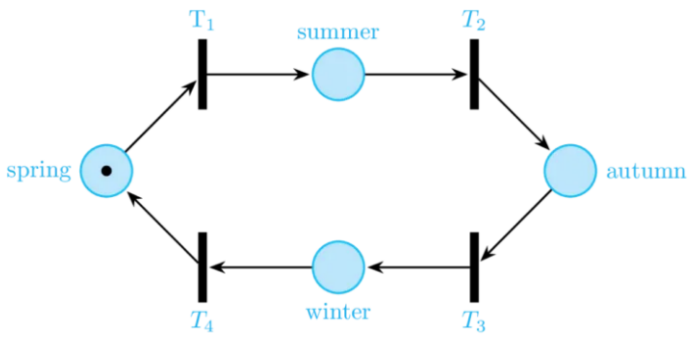


Figure 1: A simple PetriNet looping through the different seasons of the year [2].

The PetriNet in figure 1 is called seasons and gives and easy understanding to how PetriNet works. Place "Spring" starts with one token, as "T1" only have one place inputted the token will be fired and moved to "Summer", then the token will loop through all the places one by one, simulating a year.

As mentioned Places can have multiple tokens, and transitions can have multiple input and outputs. PetriNet is also non-deterministic, so in the case of a net with the ability to have more than one transition is enabled at once, the tokens will be fired in any order. PetriNet can be used to simulate multiple problems, some of the more popular are "DiningPhilosophers" and "Mutex", which both are represented in the language that is implemented and can be found in the git repository, among other examples in the project.

3 Solution

This chapter will explain how PetriNet is currently implemented in MPS. First looking at the implementation of the language structure. Then looking at what constraints are implemented, what syntax the language uses, and finally looking at how the language is executed.

3.1 Structure

This implementation of PetriNet uses a very simple structure. Each PetriNet has a number of elements of type PetrinetElement as seen in Figure 2. Each element can be either a Place or a Transition because both Place and Transition extend PetrinetElement. A Place must have some number of tokens ≥ 0. Each Transition has any number of PlaceRef as a input and any number of PlaceRef as an output. A PlaceRef refers to exactly 1 Place. All PetrinetElements including the PetriNet have a name.

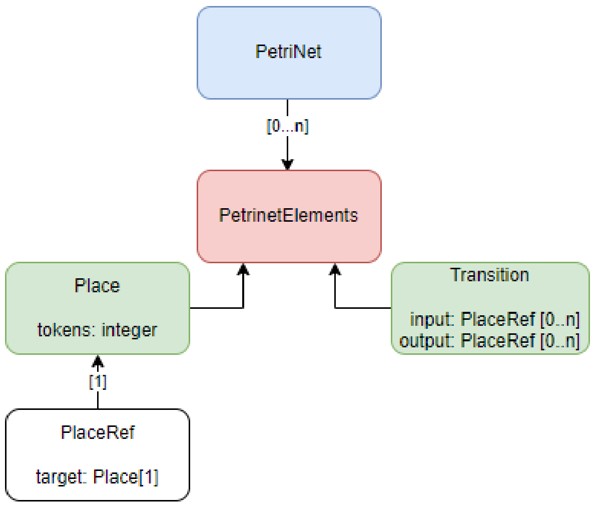


Figure 2: PetriNet implementation structure.

3.2 Constraints

There are multiple constraints implemented in MPS in multiple different ways, here we will list them:

Constraints implemented by the **typesystem aspect**:

- PetriNets cannot have the same name.
- A PetriNet cannot have the same name as a Place or a Transition inside it.
- A Place and a Transition can not have the same name.
- A Place can not have the same name as another Place in the same PetriNet.

- A Transition can not have the same name as another Transition in the same PetriNet.
- A Place cannot appear more then once in the Input of a Transition.
- A Place cannot appear more then once in the Output of a Transition.

Constraints implemented by the **constraints aspect**:

- The constraint for any name in PetriNet is given by a regex pattern "[a-zA-Z[_]][a-zA-Z0-9[_]]*". This means names must match this regex pattern to be used.
- Transitions must have PlaceRefs that refer to a Place that currently exists in the same PetriNet. This means a PlaceRef which refers to a Place in another PetriNet cannot be used.

3.3 Syntax

The language is presented as text. However the PetriNet programs in our implementation are not text files. The text representation is only a view of the underlying AST kept by the projectional editor MPS. MPS allows storing the file as XML text, but the text shown to the user is different. In the presented text each place and transition is a line of code. Currently our implementation does not have an option to display the PetriNet programs as diagram, but it should be possible to implement in the future since the programs are actually ASTs.
Here is an example of how a PetriNet is displayed by MPS:

```
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
}
```

3.4 Semantics

It is possible to run the language both by compilation and an interpreter. Therefore the semantics of the language is defined both operationally by how the interpreter reacts to a particular PetriNet program, and translationally by model-to-model translation into MPS baselanguage. More precise description is given in subsection 3.4 and Figure 3.4. The interpreter and the transformation to baselanguage is defined to be similar semantically and to fit language description seen in section 2.

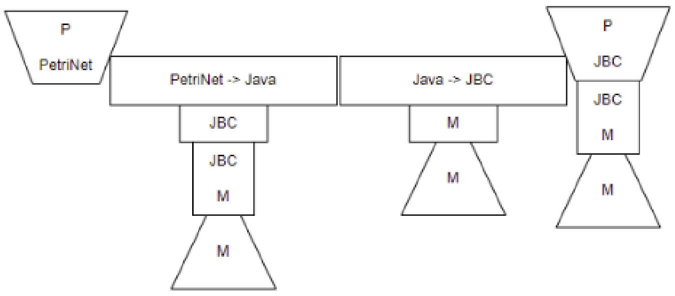


Figure 3: PetriNet program P getting translated into baselanguage to be further translated further and then interpreted.

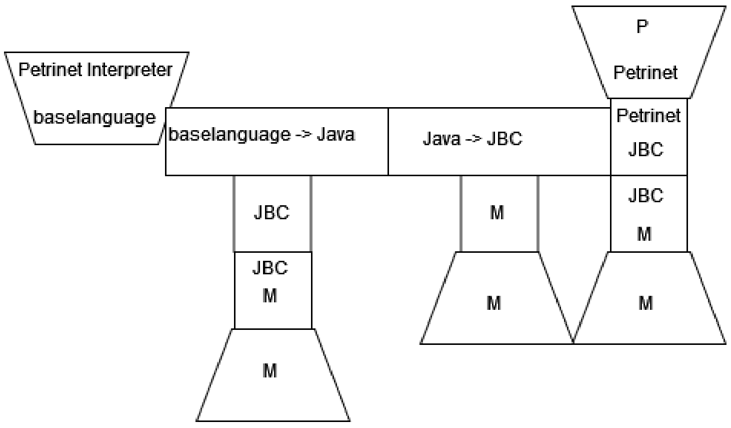
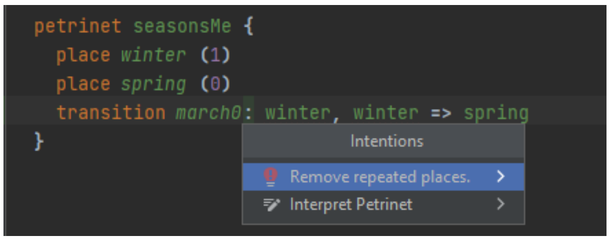


Figure 4: PetriNet program P getting interpreted directly using an interpreter.

3.5 Intentions

Currently there are two intentions implemented shown in Figure 5. The ”Remove repeated places.” intention, appears whenever there are two or more places with the same name in the input or the output of a transition. When executed it will remove all occurrences of the Place except one in either the input or output. The ”Interpret Petrinet” intention is always available and when executed will interpret the PetriNet and write to the ”message” output.



```
petrinet seasonsMe {
  place winter (1)
  place spring (0)
  transition march0: winter, winter => spring
}
```

Figure 5: PetriNet intentions.

4 Discussion

In this chapter we will explain what we have implemented and how we have implemented it. We will look at the implementation of the Intentions and constraints implementations, as well as the interpreter implementation.

4.1 Intention & Constraint implementation

The ”Remove repeated places.” intention is implemented such that it only appears when there are two or more Places with the same name in either input or output of a transition. This is done by first checking the input then the output of a transition. We iterate for each Place and check if there are more then one appearances of that Place in the input/output list, if yes we instantly return. On the execution of this intention we iterate of each Place in the Input/Output and remove all elements which have more then one appearance in the input/output. This means that when removing duplicates, the first ones in the list are the ones removed.

The constraint implemented for the names of PetriNets not being equal to the names of the Places or Transitions is implemented in the same way the Intention is implemented.

4.2 Interpreter implementation

The interpreter for in our PetriNet implementation is implemented as a method in the behavior of the main ”PetriNet” element which is the root of the abstract syntax tree.

One strong side of the interpreter over compiling the language is that we don’t need to spend time compiling the PetriNet file, the interpreter is instead compiled together with the language which happens less often than running the files. We can get an almost instant feedback after making changes to a PetriNet instead of waiting for it to be compiled into MPS baselanguage, then into Java and finally Java bytecode before being finally executed by the JVM.

Behavior methods do have some limitations. The methods of the behaviors in MPS run synchronously in the main thread, which freezes the IDE while the program runs. It also does not provide any way to stop it once it starts running, and even if it did the MPS IDE is frozen anyway. Therefore unlike the compiled version of the language it needs to fire a limited amount of transitions. We chosen 100 000 fired transitions as the limit because it takes short time, and if the user wants to run PetriNets should be ran for more, compiling the PetriNet the previous way is still an option.

5 Plan and reflection

5.1 Plan

To progress through the project the group set a goal to work at least three hours a week, in the first weeks, then work longer if needed towards the end as some courses would be completed and more time would be freed up. We also planned to have weekly meetings, either only with the group or with the supervisor, by meeting at school or online.

Time spent was the 3 hours per week as we planned, a little less when other projects were due and needed to be prioritized. This also led to working longer the last two weeks to finish tasks.

To begin with the plan was to get to know the language and look at it together, then split up the different to-do tasks between us after we gained an understanding of the language and MPS.

5.2 Reflection

The original assignment was very vague, as initially we were assigned to implement whatever that can be found in the TODO.txt file. This means we had complete freedom on what to implement. And while we did get the specific request of implementing an interpreter, overall there is not a clear objective that we can say we have accomplished. We did implement the interpreter and we did implement and solve other problems written in the TODO.txt file. However we personally feel that more could have been done, especially if the initial assignment had been more specific.

Using MPS was a bit challenging as it is very different from what we are used to. It was also a bit difficult to find answers ourselves, but once we started working with it through some trial and error, and with help from the professor, we started to get the hang of it and learned fast to use alt-enter and ctrl-space for shortcuts.

Questions for next years students:

What can the inspector be used for?

Can be used to set property values within template fragments.

What type is a variable containing a AST node whose concept type is Place?
The variable containing a AST node with Place concept type has the type node<Place>

Which Collections types are not available in BaseLanguage:

- Sequence

- List

- Set

- Map

- Stack

- Graph

- nDArray

References

[1] Wolfgang Reisig. “Petri nets and algebraic specifications”. In: *Theoretical Computer Science* 80.1 (1991), pp. 1–34. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(91)90203-E. URL: https://www.sciencedirect.com/science/article/pii/030439759190203E.

[2] admin. *It’s Time to Learn drawing Petri Nets in TikZ!* 2021. URL: https://latexdraw.com/petri-nets-tikz/.