Petri Net Documentation

Christian Meter

Abstract

This project is about developing a petri net in the function language Clojure. In this document the decisions for the design of the project are described. The main object was to stay simple and functional.

1. Architecture Decisions

For this project a generic basic approach was chosen to encapsulate the program as follows:

- The *logic* of the program is implemented in `core.clj`. Without checking for invalid input all commands are executed on the DSL.
- On top of the logic there is the *API* in `api.clj`. Here are the functions which can be used by the simulator and the GUI.
- The *Simulator* uses functions from the API to make (random) modification of the petri net possible.
- The *GUI* visualizes the current DSL.

1.1. Controller

Logic All functions are here to initialize and manipulate the petri nets are in the Controller. All other functions are using these functions to manipulate the state of the program.

No inspection for valid input To keep the functions simple, none of them will check any input. They only do what has been typed into the REPL. So the liability totally depends on the user's input. That is okay, because later on will be an API available to manipulate the DSL including an inspection for valid inputs.

Simplicity Nearly every function was built simple. Only `merge-net` appears a bit complected, but it just calls all the little merge-functions and combines every result to get the new merged net.

Manipulating State Only few functions are needed to manipulate the DSL. Because there must be a place where the database of petri nets is modified, the functions are reduced to a minimum.

Data Structure For the database was a big hash-map chosen. Every net is a keyword and has as the value again a hash-map to store all information. Those information are:

- Edges, read as: `{:from {:to :costs}}` and divided into
  - from one place to multiple transitions
  - from one transition to multiple places
- Places, a hash-map containing name of the place and number of tokens
- Transitions, a hash-set containing the names of the transitions
- Properties, a vector including the properties defined in the Simulator.

So an example net will look like:

```
{:example-net
 {:edges-from-trans
    {:from-p {:to-t :cost}}
  :edges-to-trans
    {:from-t {:to-p1 1 :to-p2 1}}
  :places
    {:p1 0 :p2 1}
  :transitions
    #{:to-t :from-t}
  :props
    [(petri-net.simulator/net-alive?
                          :example-net)]}
 :next-net
 ...
}
```

Choosing this structure provides the great ability to use the function `assoc-in`, which makes most of the work when the user wants to create or modify one of the edges. The simple call

```
(swap! nets assoc-in [net :edges-to-trans from to] tokens)
```

has two functionalities:

1. Update existing edge
2. Add new edge

And this is exactly that, what is needed for this structure.

1.1.1. Properties

The properties are here stored via quoting into the vector. When the property is evaluated, the result is printed as `true` or `false`. With this method it is possible to evaluate the properties when they are needed and not as the user adds them to the DSL. Everytime something was changed in the database, the result of the properties is displayed in the GUI.

The datatype vector was chosen because it provides a good method to store function calls into a data structure, which does not execute it directly and which is no list. Otherwise there could be problems with trying to evaluate a boolean provided by the functions `net-alive?`, `transition-alive?` and `non-empty?`.

1.2. API

If possible return value, else nil In the API are functions built in which check the user's input and executes it if possible. If not, it will always return `nil`.

Built on top of the Controller Every function needed to directly manipulate the DSL, is provided by the API. So the user does not need to get into the pure logic of this program to work with the petri nets.

The API was designed to move the verification of the user's input into another place than the core. So this job could be encapsulated from the core.

1.3. Simulator

JavaFX vs. SeeSaw

1.4. GUI

JavaFX vs. SeeSaw