Motivations and Design behind Agar

Alexander Schneider[*]

Heinrich Heine University Düsseldorf
alexander.schneider@hhu.de

Abstract

*This paper details the reasons and motivations behind the petri-net Simulator Agar. Agar is capable of creating and simulating place/transition petri nets. Because Agar is written in Clojure one of the main design paradigms was to keep Agar **Simple**.*

I. Introduction

Clojure is a programming language which has simplicity as one of its main paradigms. Following this tradition we introduce Agar, a simple place/transition petri net simulator written in Clojure. Simple programming tries to minimize the interactions of a method. An implication of this is to minimize state, because state causes unknown and uncontrollable interactions. With simplicity in mind Agar is divided into several Clojure files which fulfill a different role described in detail in the following chapters.

II. Why Clojure?

As mentioned before the main paradigm Clojure is designed after is simplicity. For reference we can compare Clojure and Java for the purpose of creating an Petri net Simulator. In Java it would be a good Idea to use Interfaces and Object Oriented Programming as main guidelines for Programming. The State (mainly the State of the nets) could be handled in the form of mutable Objects. One could design a net Object and let a Net-Collector Class handle the nets and the operations on them. The problem with this approach would be that statefull objects are introduced as components of a statefull class. The immutability of Clojure lends itself to the avoidance of state. All state can be handled in one atom and is therefore bundled at one place, further simplifying the matter.
This and the fact, that Clojure is simpler than Java by design are the main Reasons to use Clojure. Every Operation that should per Chance not be possible in Clojure could be handled trough Java Interop.

III. Data Structure

The Data structure which is used to represent and save the petri-nets was developed, before the programming and the design of Agar itself began. Only the property part changed during implementation and is described further in the Refactoring chapter.
The entirety of all nets are represented as a hash-map. The hash-map contains the net-names as keys and the nets as values. A single net is a hash-map as well with the keys :places, :transitions, :edges-in, :edges-out, and :props. The value of :places is in turn a hash-map which holds the name of the place as a key and the number of tokens currently in the place as a value. The value of :transition is a set, that contains all transition-names. The values of :edges-in and :edges-out are also hash-maps. The keys of those hash maps are the transitions where the edges are going in or out of respective. The values there are in turn (you probably guessed it right) hash-maps with the places where the edge is coming from or going in respectively as keys and the cost of the

[*]All praise RNGeesus

edge as values. The value of :props is a set containing the property-strings. An Example data-structure looks like this:

```
{"foo" {:places {"baz" 5, "bar" 5},
:transitions #{"bam"},
:edges-in {"bam" {"baz" 1, "bar" 1}},
:edges-out {"bam" {"baz" 1}},
:props #{"(petri-nets.simulator/netalive "foo")"}}
 "test2" {:places {"fizz" 9, buzz "10"},
    :transitions #{"fizzbuzz"},
    :edges-in {}
    :edges-out {}
    :props #{}}}
```

IV. Functionality

Agar can be used via REPL using the simulator.clj or via GUI using the gui.clj. Both librarys have the almost the same functionality. It is possible to create nets, add places, transitions and edges and properties. Nets can also be deleted, copied, and merged. Additionally user-selected or random transitions can be fired.
The GUI also auto-evaluates all properties after a transition has been fired. This function was omitted in the simulator, because in the REPL thats just one call to `(eval (read-string))`. Firing of a random transition x times is also omitted in the Simulator. Again because it is just a simple call of `(times)` on the random-fire function.
Every Net can be saved or loaded separately in addition to saving or loading the whole database of nets.

V. Implementation

Agar is divided mainly into 4 parts.

- Core
- API
- Simulator
- GUI

I. Core module

The `core.clj` is the heart of the application. Its sole purpose is the manipulation of the domain specific language. For example creating new petri nets, adding and deleting nodes and edges, saving and loading nets, etc. The core does not check whether the constructed petri nets are valid or not, it simply executes the Functions. Because of this every function either just has to execute one simple set of instructions or is a composition of aforementioned methods. Furthermore the core manages a database of nets inside an atom. The `merge-nets` function is the only one which could be considered not simple. But since it only acts as a means to "bundle" other functions it should be acceptable nonetheless.

II. API

The `api.clj` encapsulates the core. The main function of the API is the filtering of the input to prevent faulty petri nets. Every function that involves user input checks whether this input is valid (e.g. adding place with a name already present in the net is not valid). When the input is valid the function just passes the call to the corresponding core function.

III. Simulator

The `simulator.clj` is one possible application/use case of the API. It takes the abstract of a petri net that the API provides and uses it for the concrete case of a simulator.
The simulator provides properties that can be added to the petri net. Furthermore the ability to "fire" a transition is provided. The simulator uses the API functions to change the state of the net to simulate the execution of a transition.

IV. GUI

The `gui.clj` provides a graphical interface for the simulator. This is the point where the simplicity breaks. This file heavily utilizes the `seesaw` library. Seesaw is a clojure library that builds on the Java Swing Toolkit. Swing does not comply with simplicity guidelines. While using swing there is no way to separate the code for the appearance and the code which fulfills a function.

While seesaw helps make swing simpler it can not change the core design of swing. In the case of Agar we tried to first design the looks of the GUI and then attach the functions trough listeners in the main Function. This results in the main function being the only point where function and appearance are intertwined.

VI. Refactorings

Due to previous planing described in Chapter II there was only one big refactoring in the design of Agar. The handling of the properties changed during the implementation of the GUI. Before that the properties were saved as a sequence which evaluates to a function call. Because of the nature of the GUI and the decision to give the user the ability to operate on all possible properties with `or` and `not` through mouse-clicks the representation changed. The new representation was a String representing the sequence that evaluates to a call executing the property functions. Because of that all property handling functions in the core had to be adapted to the new format.

VII. Verification

The verification of the code was not done formally. To elaborate on the reasons for this: Rich Hickey, the creator of Clojure, stands on the point, that simple and well designed programs do not need to be tested. (For further elaboration see his talk "Simple made Easy"). It is our opinion, that agar fulfills both properties of being well designed and simple. The only not simple point, the merge function, was tested extensively through a lot of applications. Because merge-nets acts as an aggregator as mentioned before, formal testing is not needed in our opinion.