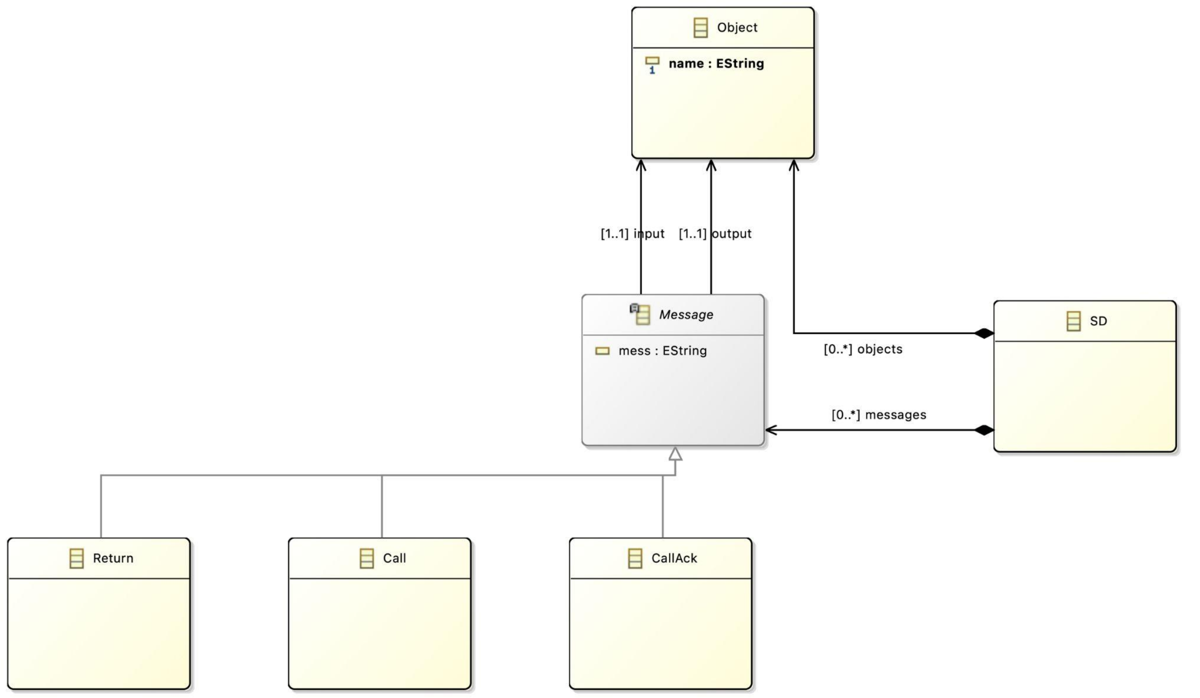
# UML Sequence Diagrams And Petri Nets Interoperability Project

**Mykola Fedurko**

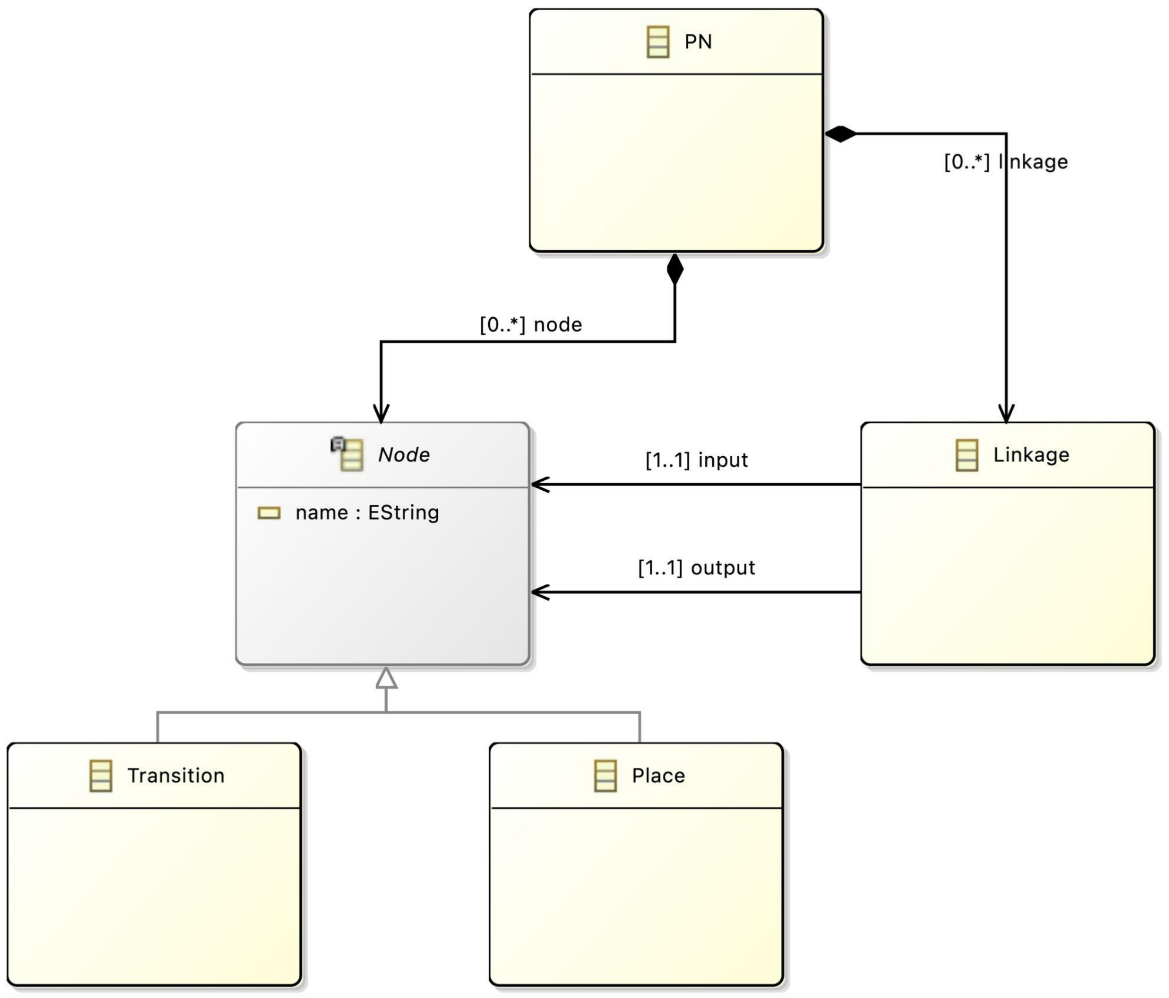
## INTRODUCTION

In this project, we are transforming the UML Sequence diagram into Petri nets. First of all worth to note that we decided to limit the UML Sequence diagram to the essence of its mechanics because the purpose of the project is to just get Sequence diagrams and Petri nets interoperable. Take a look at the Sequence diagrams metamodel below:



Picture 1. Sequence diagrams metamodel

In the meantime, Petri nets is itself a pretty minimal modeling language(partly because of its mathematical origins). So its metamodel is self-describing as was stated in the previous paper about the design of the metamodels.



Picture 2. Petri nets metamodel

In this part of our project description, we are aiming at describing the transformation. So above is just a brief reminder of what exactly we are going to transform.

From now on in this paper, the PN stands for Petri nets and SD stands for Sequence diagrams.

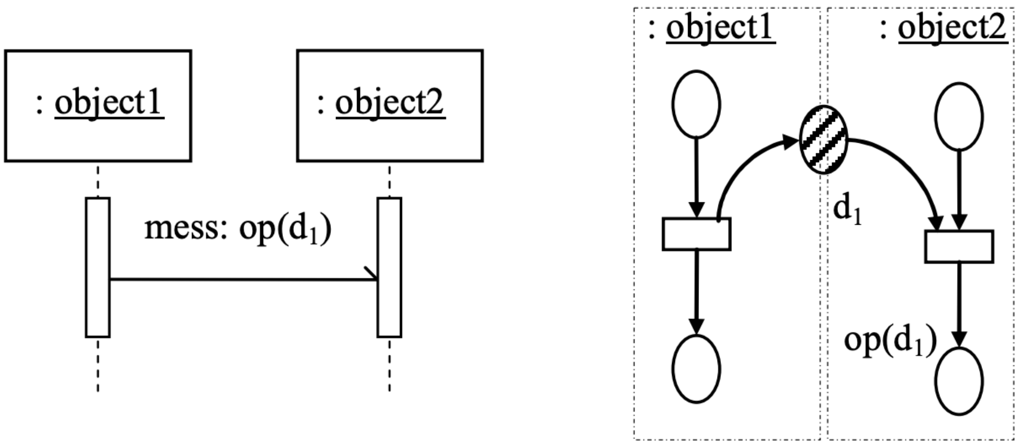
## ALTERNATIVES AND JUSTIFICATION OF CHOSEN SOLUTION

Basically, we had not had many tools to choose from due to the novicity of MDE to authors. But the variants for consideration were:

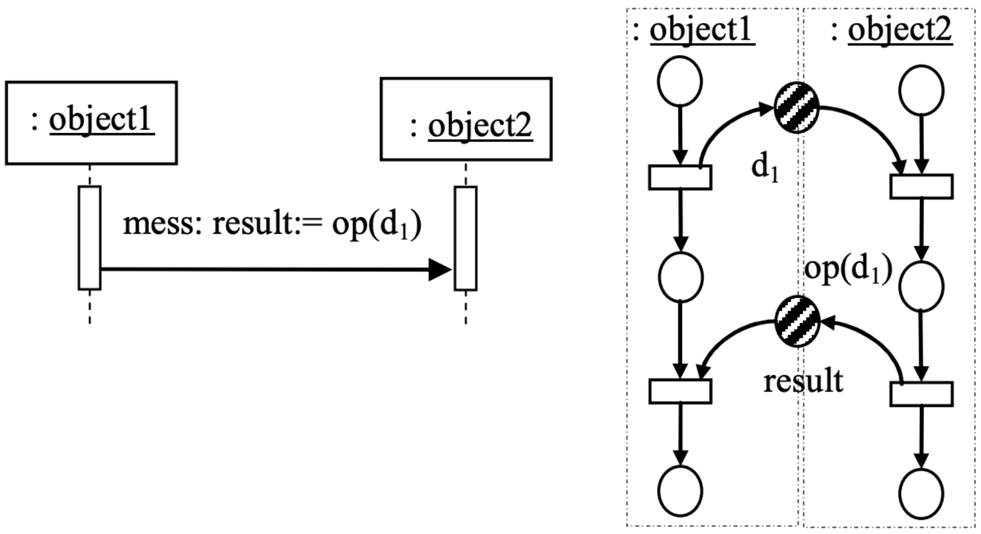
- ATL
- QVT

At first, our favorite was ATL because of its declarative nature and simple structure. But the closer we were getting to implementation the clearer it was that it would be much easier to use more complex first-glance language to describe some complex transformations.

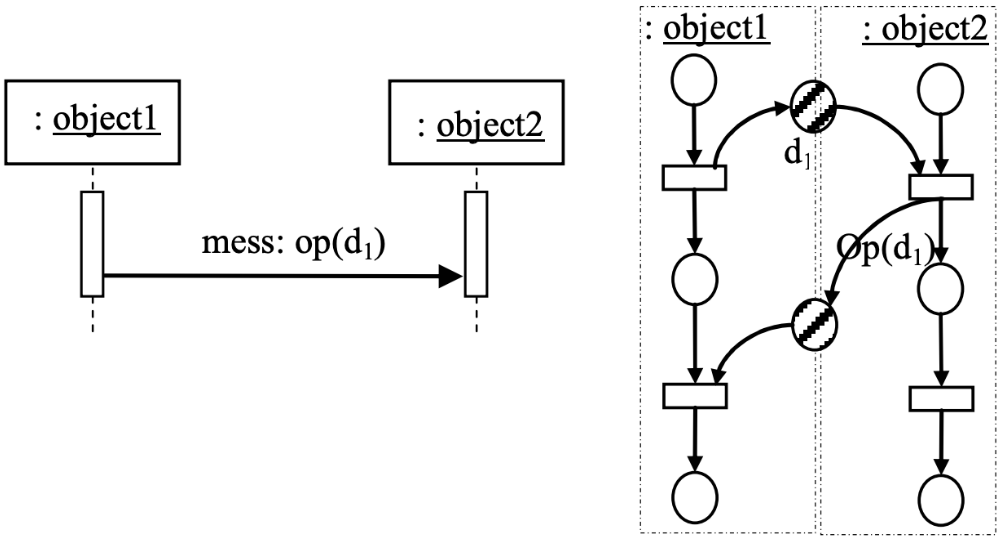
For example, as you may remember, we had three types of messages in SD: Call, Return, and CallAck. All of these were going to be transformed into specific Petri net patterns[1]:



Picture 3. “Call” transformation



Picture 4. “Call” and “Return” transformation



Picture 5. “CallAck” transformation

After all of these are transformed their sequence should be connected by transitions to form a sequence of Petri nets(resulting in Petri net). So the approach was clear: sequence of messages to resulting Petri net consisting of smaller Petri nets of corresponding message types. And it is these transitions between them that were the problem with ATL.

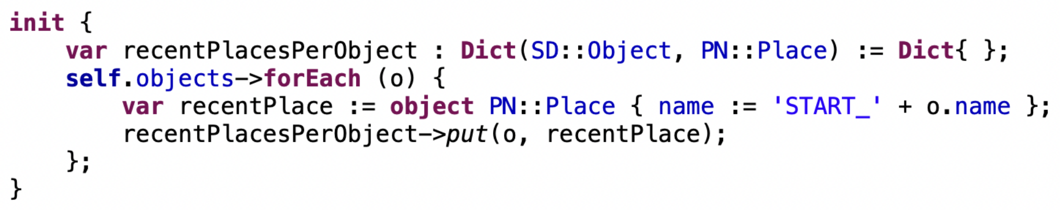
Now we can’t just transform single message types to appropriate Petri subnets in a single rule. Now we have to also run through all messages in a loop transforming them, and in the meantime connecting them with needed transitions. Besides, as we know, ATL rules are pretty nondeterministic in their order of execution, so we can't just write rules to transform them in any order. The order and the state of messaging in the Sequence diagram are crucial.

Running a loop for all messages is already a task complex enough to consider changing the instrument, or at least the approach. It was decided to do the former and switch to a well-known accustomed instrument: QVT.

QVT provides a much more solid inventory to choose from and lots of possibilities for tricks. It has mappings, helpers, intermediate classes, flow constructions, parameter modifiers, multiple returns, and so on.

## SOLUTION DESIGN

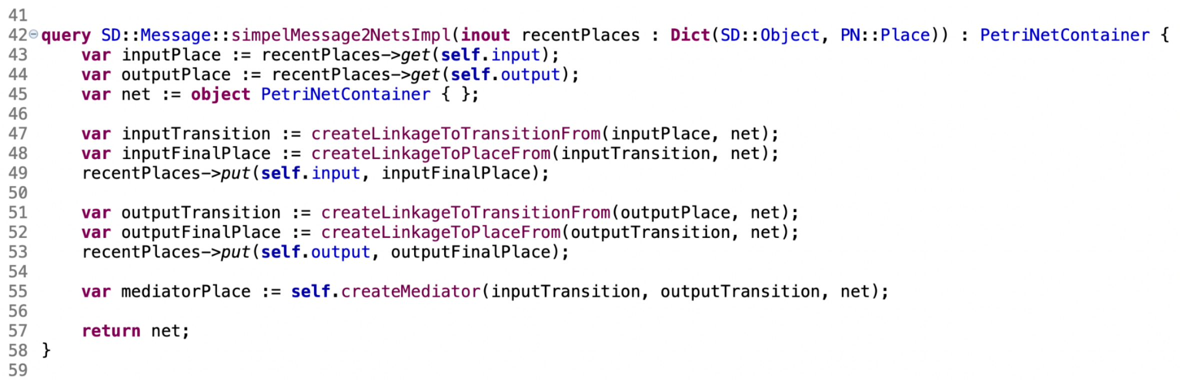
First of all, we initialize the dictionary of the initial Place per every object in an SD. After that, as was stated above, we are just iterating over all of the messages in an SD and getting Petri nets generated for them. When generating PN for a message from Object1 to Object2 we are taking into account its current last place. At the start it's mainly "START_Object1" and "START_Object2" respectively, which are generated like this:



```
init {
    var recentPlacesPerObject : Dict(SD::Object, PN::Place) := Dict{ };
    self.objects->forEach (o) {
        var recentPlace := object PN::Place { name := 'START_' + o.name };
        recentPlacesPerObject->put(o, recentPlace);
    };
}
```

Picture 6. Generation of initial places

After that, we are running the transformation rules per each Message type in Pictures 3, 4, and 5. The dictionary "recentPlacesPerObject" is being passed to all of the rules and every two objects taking part in communication are having their last places changed via this dictionary.



```
41
42 query SD::Message::simpelMessage2NetsImpl(inout recentPlaces : Dict(SD::Object, PN::Place)) : PetriNetContainer {
43     var inputPlace := recentPlaces->get(self.input);
44     var outputPlace := recentPlaces->get(self.output);
45     var net := object PetriNetContainer { };
46
47     var inputTransition := createLinkageToTransitionFrom(inputPlace, net);
48     var inputFinalPlace := createLinkageToPlaceFrom(inputTransition, net);
49     recentPlaces->put(self.input, inputFinalPlace);
50
51     var outputTransition := createLinkageToTransitionFrom(outputPlace, net);
52     var outputFinalPlace := createLinkageToPlaceFrom(outputTransition, net);
53     recentPlaces->put(self.output, outputFinalPlace);
54
55     var mediatorPlace := self.createMediator(inputTransition, outputTransition, net);
56
57     return net;
58 }
59
```

Picture 7. "Call" transformation implementation

For example in Picture 7 we see that:

- "input place" and "output place" are taken from this dictionary on lines 43-44

- being manipulated and generated on lines 47-55
- but after all of these final “clue” places are saved into dictionary replacing older final places per object on lines 49 and 53

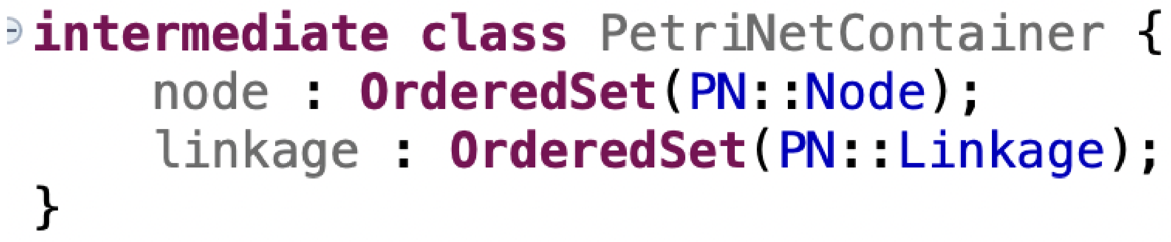
By the way, we encapsulate basically the same transformation for SD::Call and SD::Return in a simpleMessage2NetsImpl.

Also, we have helpers:

- createLinkageToTransitionFrom
- createLinkageToPlaceFrom
- createMediator

that help to hide the clutter of these manipulations with places and transitions.

As a final note, we use the “PetriNetContainer” intermediate class to store some variables. It basically mirrors the structure of PN::PN class for interchangeability.

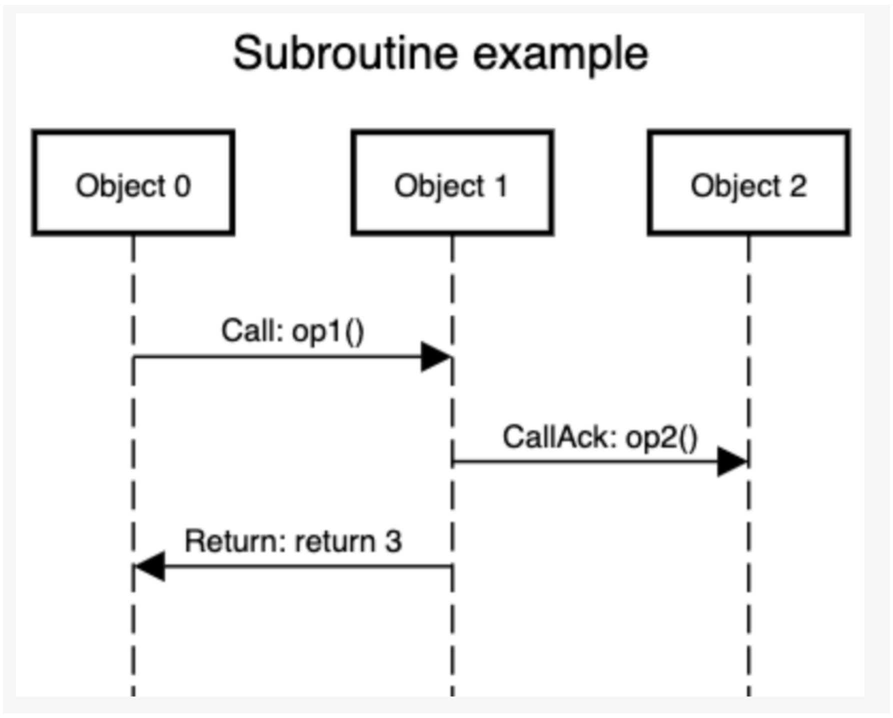


```
intermediate class PetriNetContainer {
    node : OrderedSet(PN::Node);
    linkage : OrderedSet(PN::Linkage);
}
```

Picture 8. PetriNetContainer

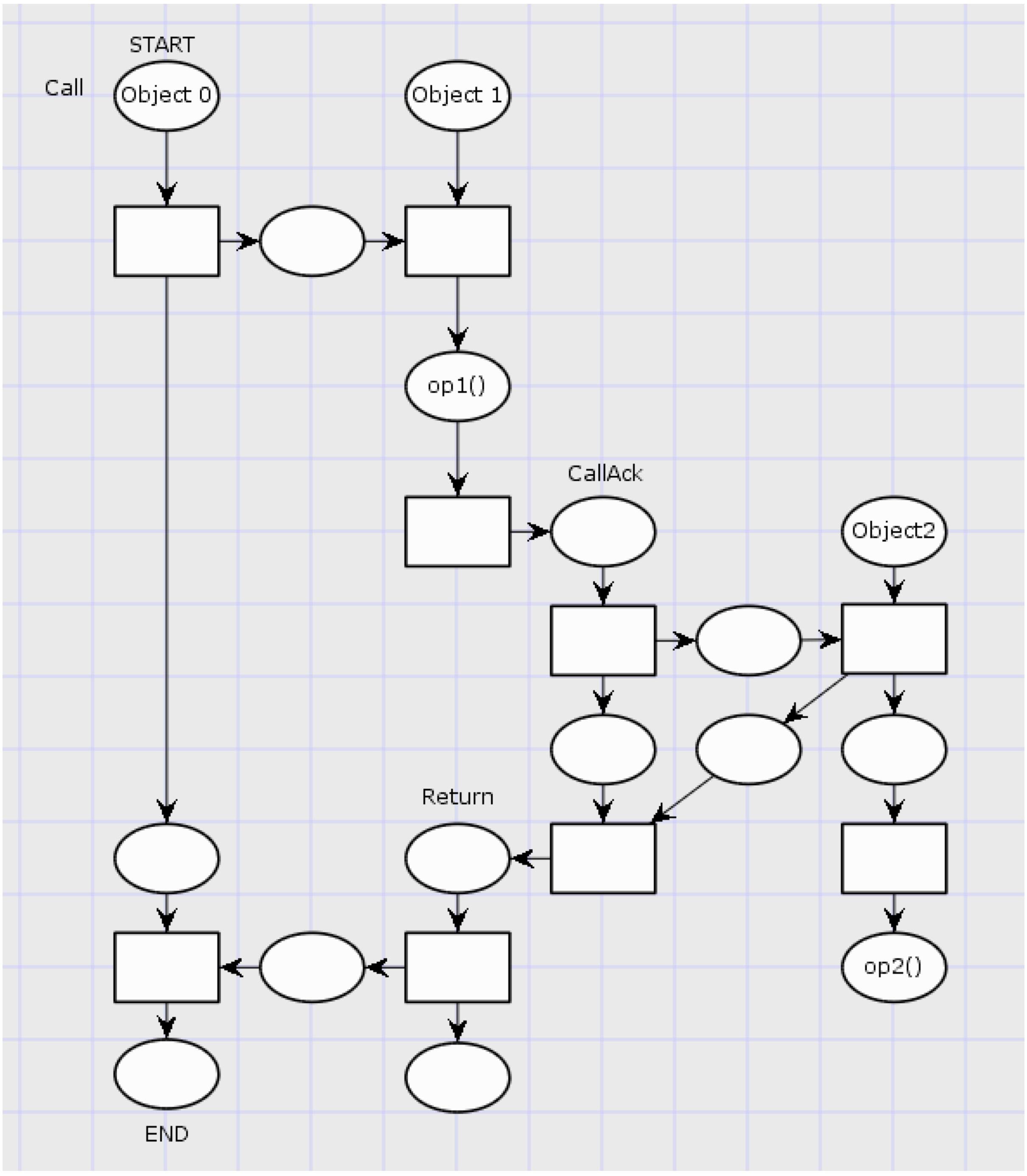
## TESTS AND EVALUATION

Let’s try to transform the Subroutine example from the previous Project task description:



Picture 9. Subroutine sequence diagram

It should be transformed into the following Petri net diagram:



Picture 10. Subroutine Petri net diagram

Text labels are given just for descriptive purposes. If one will take a look at Pictures 3-5, it will be immediately clear how this works. It's just different types of messages glued together by a few transitions.

Probably it would be more suitable to give more descriptive labels to each of the transitions/places so that users could read them more intuitively.