Pan-European University, Faculty of Informatics

Interim Research Report

Definition of inter-process communication

of the project

Requirements and formal definition of a low-code language based on object-centric processes - LowcodeOCP

Authors:
Gabriel Juhás, Milan Mladoniczky, Juraj Mažári and Tomáš Kováčik

Contact:
gabriel.juhas@paneurouni.com, milan.mladoniczky@paneurouni.com

Funded by the EU NextGenerationEU through the Recovery and Resilience Plan for Slovakia under the project No.

09I03-03-V04-00493

October 1, 2025

Version: v0.1

Abstract

In modelling of business processes, a process is mostly considered as independent, self-sustainable entity, that works only within its boundaries. But in reality, every activity is connected in some way to other activities. For this purpose, the modelling language Petriflow [M M17] introduces inter-process synchronization or also called process communication. This paper describes fundamentals of influencing one process from another one and usage of this concept for implementing means of communication between instances of the same process.

This interim report summarizes the current progress of the task 2.2 of the research project LowcodeOCP resulting in definition of inter-process communication.

Contents

List of Figures

Chapter 1

Introduction

Business process modelling, also known under the abbreviation BPM, is ever growing discipline both in the commercial and academic sphere. Nowadays, there is a large number of modelling languages and tools for them. One of which are Petri nets [J D01]. Petri nets are the perfect tool for modelling processes in every domain. Thanks to their simple but expressive nature, they are widely used for creating easy-to-read models of complex processes. A Petri net is a directed bipartite graph, which nodes are represented as transition and places. Transitions of the net represent activities or events in a modelled process. Places, on the other hand, describe conditions for process events to happen. A Petri net nodes are connected with directed arcs. It is a rule in Petri nets, that an arc can be drawn only from a place to a transition or from a transition to a place. Arcs directed from a place to a transition are called input arcs for the transition. In a similar manner, arcs directed from a transition to a place are called output arcs.

The main advantage of Petri nets and the reason why they are suitable for expressing complex processes is the option to simulate the execution of a modelled process. Every Petri net has a marking that describes a state of a process model. The marking is represented with a number of tokens in net's places. Each transition of a net can be executed, or as called in Petri nets, fired when in all input places of the transition is a sufficient amount of tokens. A transition that can be fired is called enabled transition. When a transition fires, tokens from all input places are consumed and produced in all output places of the transition according to connecting arcs. With Petri nets, various complex processes or procedures, with parallelly executing activities, can be modelled.

In original Petri nets formalism, every model operates only within its own space. In the real world, one or more events outside of the modelled process can influence the behaviour of the process. In Petri nets, it can be described as if one process acts as an actor in another process. With this concept in mind Petriflow modelling language extends Petri nets by elements to define the desired behaviour of modelled processes. In Petriflow it is possible to model a group of processes that communicate with each other.

Chapter 2

Petriflow

Petriflow is a modelling language based on Petri nets. It builds upon features of Petri nets and extends them to provide more tools for modelling complex processes. In existence of Petri nets, there are numerous extensions of the original formalism, such as CPN [M B01] based on Coloured Petri nets [CW 05], Viptool [R B06], [JN03], Yasper [KM 08] or ProM [B v05], to mention just a few of them. Most of the tools, for Petri nets or its extensions that are available today, are determined to create models and some of them to analyse the models. Petriflow was created with another purpose in mind. Its main goal is to provide enough tools and components to define a process that can be executed as a fully functional application. Petriflow as a language blurs the border between modelling and programming languages. Petriflow uses advantages of a modelling language, such as an easy-to-understand form of a graph, mathematical definition, option to analyse and simulate models before its deployment, and applies it to create software applications as a programming language. Like with every programming language, it needs a compatible language interpreter or application engine to utilize and execute Petriflow models.

To enable creating a deployable process models Petriflow extends Petri nets formalism with several components. Firstly more types of arcs are introduced such as inhibitor arc, reset arc, and read arc, that are well-known extensions of Petri nets. Next, the firing of a transition is enriched with more steps to better describe the behaviour of an activity which a transition represents. A transition with a more complex act of execution is in Petriflow called a task. A task consists of five events that may or may not be executed:

- **Assign event** - consumes tokens from input places of the transition
- **Action** - the state of the transition where an actor executes activity represented by the transition
- **Cancel event** - the execution of the task is cancelled and consumed tokens are returned to the input places of the transition

- **Delegate event** - the optional event which delegates execution of the task to another actor
- **Finish event** - produces tokens to output places of the transitions

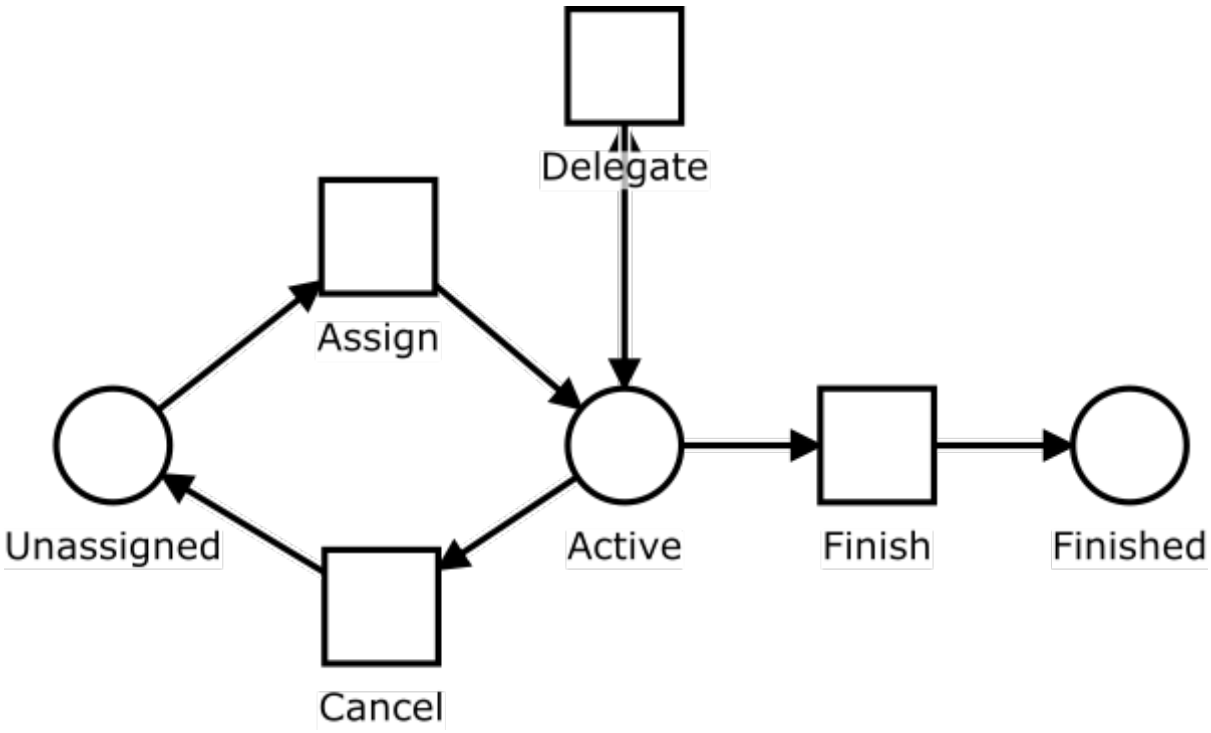

Figure 2.1: Petri net defining events of the Petriflow task transition [M M17]

In Petriflow, an actor is an entity that can fire a transition or a task. An actor can be understood as a user of the system or in case of automation a system itself. To cover the most of the needs to build an application process Petriflow next extends Petri nets with roles, data and so-called actions.

Roles in Petriflow provide means to authorize chosen actors to execute certain transitions or tasks. If a role is assigned to a transition, only an actor with the specific role can fire the transition.

Model data are variables to store values needed for the desired behaviour of a modelled process. Data variables can be of various types, for example, but not limited to, text, number, date, boolean, enumeration etc. Every data variable defined in a process model can be bound to a transition where complementary definitions about variable's behaviour within the transition are added. Data variable with behavioural characteristics within a transition is called a data field.

The last, but not least, extension of Petri nets are actions. Actions are small snippets of code that defines a reaction to a certain event within a transition or a data variable or a data field. They can be understood as functions that run every time when an event, inside a Petriflow component to which they are bound to, happens.

As mentioned before, the main goal of Petriflow language is to create deployable process models, so Petriflow models are exported in XML format and embedded into a compatible application engine to create a fully functional application. The only exceptions in a Petriflow model are actions. They are written in the domain-specific language based on the programming language Groovy DSL.

2.1 Multi-process environment

If we imagine modelling of a process, we typically mean creating a model of some process that consists of a number of activities, in Petri net transitions, connected with trigger conditions, in Petri net called places. In a standard way of modelling a process in Petri nets, we do not take the outside environment of the process into consideration. So, the modelled process is unaware of its surroundings or the context that it is situated in. However, modelling processes in Petriflow language for the purpose of building an application, cannot be done this way. Applications that are written in Petriflow language do not contain only one process. A typical Petriflow application is created with mostly two or more processes. Each application process represents a functionality or a feature of the deployed application.

An application process can be modelled to describe life-cycle of an entity that exists inside the system or to express a certain functionality that is desired in the application. For better understanding, we introduce the example of an application for a restaurant. The manager of the restaurant wants a system to enable table reservation and to monitor various activities of restaurants waiters, for example, communication with customers. After some analysis of the situation in the restaurant, we decide to model processes which create the desired application. First to be created are processes of the entities that exist in the context of the restaurant. Processes of life-cycle of a table, a customer and a waiter are modelled. Next, we want to be able to reserve a table in the restaurant, so the process of table reservation is introduced to the application. With enough knowledge of the inner workings of the restaurant and after consultation with the restaurant's staff, the desired application can be created with these four processes.

As you can see from the example, there is one big problem. If we model the processes as separate components of the application, we cannot guarantee the desired behaviour of the application. Because the process of a reservation does not know about the process of a customer, it cannot pair a table reservation activity with the right person, and the process of life-cycle of a customer is never notified, that it has a reservation. The same situation can be seen between processes of a table and a reservation. Someone can argue, that this problem could be easily solved by manually pairing the customer process to the reservation process. However, that is not the desired solution as we are trying to automate as much as possible and let users of the system focus on their own tasks. In

this example, it is clear to see why inter-process communication is needed for building rich applications from processes.

2.1.1 Infinite instances of the process

The majority of restaurants have more than one table, one waiter and one customer. How can we model more tables if we have one process of a table? In Petriflow, deployed process model serves as a template. It defines all activities (transitions) in the process when the transitions can be fired and also who can fire the transitions and what data are bound to them. The deployed process model does not contain a marking of the process or the actual values of the data. With every new table in the application, a new instance of the process is created. An instance of the process is also called a case. When it is desired to create a new instance of a process, for example, adding a new table to the restaurant, the whole model of the process is copied and the special transition in the process is fired. The transition is called a constructor and every Petriflow model begins with one. The constructor produces tokens into to places according to the initial marking defined in the process model. The constructor also initializes default values in data variables if the data variable has defined one. When the constructor is finished the new instance is ready to operate according to the process model.

2.1.2 Application scopes

As we can see, an application that is written in Petriflow language, is composed of several components, like numerous processes and their infinite number of instances. Every component of a Petriflow application operates within its own space, named scope. Petriflow defines four types of scopes:

- **Application scope** - contains all processes that create an application. It is often referenced as the global space.
- **Process scope** - concerns one process and contains all instances that were created from the process. In this scope, the deployed process model is also located.
- **Instance scope** - is the scope of one particular instance of the process. This scope contains the current marking of the process model, all data variables values.
- **Transition scope** - represent the space of one active particular transition in the instance of the process. It contains information about data fields and its states in the case of a task transition.

The scopes in Petriflow can be also illustrated as a system of layers, like in the Figure 2.2.

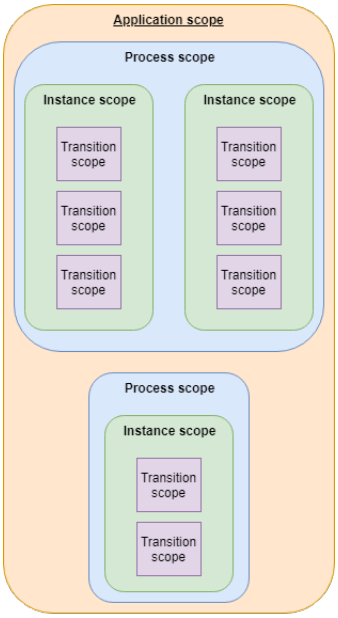

Figure 2.2: Scopes of a Petriflow application

2.2 Transition synchronization

To further explain why Petriflow can be modelling and programming language at the same time, we can use an analogy of object-oriented programming languages. You can imagine a process model as a class and an instance of the process as an object of that class. To more develop this analogy, transitions in Petriflow net can be expressed as methods of the class and places as control variables of the class. Since a method of a class is a simple function, that can be called if we have a reference to the object of the class, it should be able to call transition in a similar manner. Modifiers of methods, in object-oriented programming language, define access rights to the method such as public or private. For simplicity, we omit this attribute of the programming language in Petriflow.

Based on the mentioned comparison with the object-oriented languages, we defined means to fire transition remotely from one instance to another. This feature is in Petriflow called a transition synchronization. The basic idea is firing more transitions across different instances in a synchronous manner. In Petriflow, we have two types of transition to synchronize. The standard type of a transition, from original the Petri net formalism, called a event transition and a task transition with a more complex firing mechanism. When a transition wants to call other transitions synchronously, it has to check if chosen remote transitions meet two conditions:

1. All the chosen transitions have to be enabled within their own instance.

2. An actor has to have a valid role according to the roles defined in the chosen

transitions.

Only if all the chosen transitions fulfil these conditions, then the caller transition can fire. If it is synchronization between event transitions, all called transitions fire at the same time as the caller transitions, thus all tokens from the input places of the involved transition are consumed in their instances and at the same moment, tokens are produced to the output places of involved transitions. Synchronization of the task transitions contains more steps than synchronization of the event transitions, because of their more complex inner structure. Every event inside the task transition has to be synchronized. If an actor assigns the caller task transition, all the chosen transitions are assigned to the actor. After assigning all the chosen transitions, the actor has access to all data fields inside the transitions. If the actor is a user of the application all the data fields are displayed under the caller transition, so he can work without interruption. All other events in the task transition are synchronized as well. For example, the delegate event changes the actor in all the chosen transitions and the finish event produces tokens in all involved instances.

A transition can also call chosen remote transition asynchronously. An asynchronous call is executed regardless of enabledness of the called transitions. The second condition involving process roles still remains. The caller transition continues its firing procedure according to its own instances regardless of the call of the remote transition. This option can be useful if we want to notify other instances and we do not care if a sent message is received.

2.2.1 Selection of the synchronization partners

Until now we explained that the caller transition synchronizes with some chosen transitions. Because we cannot predict how many instances of a process will exist in a deployed application, transitions that we want to synchronize cannot be defined in a process model statically. The decision-making of synchronization partners has to be on the run-time of the application. To get the desired transition to the synchronization procedure we introduced filters to Petriflow language. The filter defines criteria for the transitions which we want to synchronize with. The filter is composed of two parts:

1. process query
2. selection

Process query is written in Petriflow Query Language. It serves to search for the desired entity, like instances, transition etc., in a Petriflow application. The query language is described in more details in the published Petriflow language definition. The selection part of the filter defines the choice of the synchronization partners, thus which transitions from the query result are chosen to synchronize with. The selection has several parameters

that define the logic of the choosing the transitions. Range parameters are the first, with values for a minimal and maximal number of chosen transitions. The second parameter is who will make the choice. The choice can be done manually when a user of the system, who fires the transition, decides which transitions from the query results call remotely or the choice can be done automatically. If the choice is left to the system, it will try to select as much as the range parameters allow.

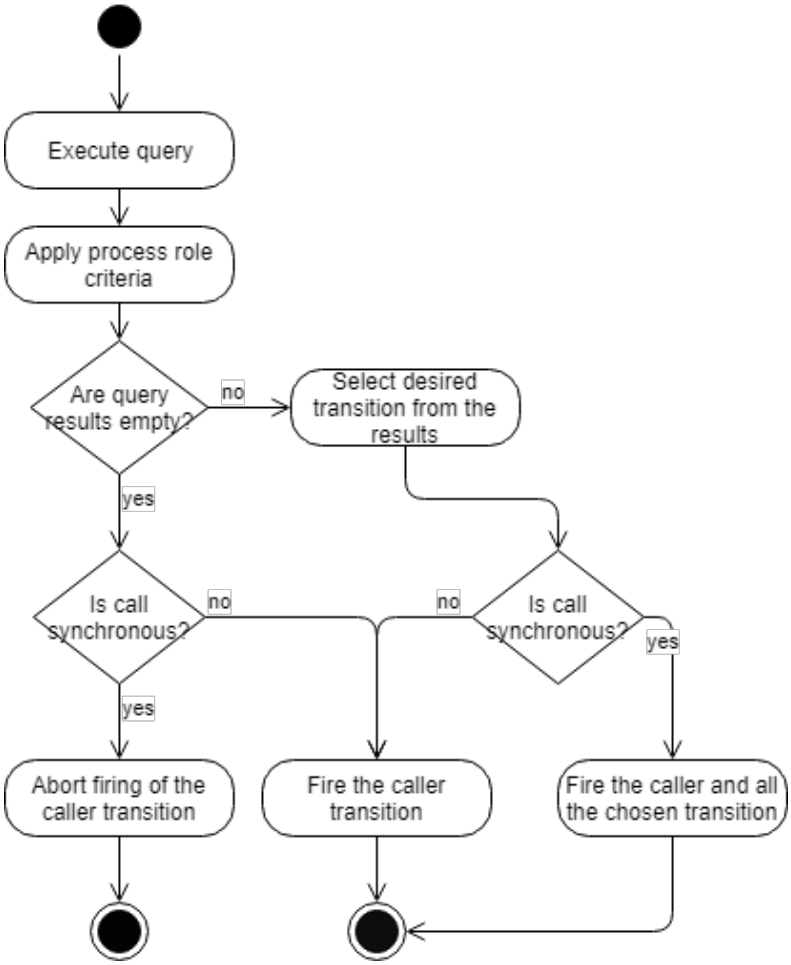

Figure 2.3: Diagram of the transition synchronization

The whole synchronization of the transitions if written in Action DSL of Petriflow language. The action defining the synchronization call is bound to the appropriate event of the transition.

The example in the Figure 2.4 demonstrates synchronization of two event transitions. The caller transition is from the customer process and the called transition is from the waiter process. As we can see when the transition 'Call a waiter' is fired by a user of the application, the action is executed. In the action, the query returns all instances of the process waiter where the transition with the label 'Customer is calling' is enabled. Then the system automatically selects one instance, thus one waiter, and on the selected instance fires the transition with the label 'Customer is calling'. Both the customer

```
<transition>
    <id>25</id>
    <label>Call a waiter</label>
    <event type="fire">
        <action>
            get cases of Waiter where that.transition['Customer is calling'] is enabled >>
            select auto min 1 max 1 >> call selected.transition['Customer is calling']
        </action>
    </event>
</transition>
```

Figure 2.4: The example of the synchronization action

instance and the waiter, with this act of synchronization, know that the waiter is coming to the customer.

2.3 Summary and outlook

In this report we provide concepts and definition of inter-process communication between instances of several object centric processes in low-code language Petriflow. Basically, we discuss relationship on data level, synchronization of events on workflow level and concept of forms as subforms on user interface level. We plan to rework the definition to provide precise formal semantics analogous with the definition of singleton object-centric process.

Bibliography

[B v05] et al. B. van Dongen A.K. Alves de Medeiros. “A New Era in Process Mining Tool Support.” In: *Application and Theory of Petri Nets 2005, LNCS 3536, pp. 444454. Springer-Verlag, Berlin* (2005).

[CW 05] W.M.P. van der Aalst C.W. Gunther. “Modeling the Case Handling Principles with Colored Petri Nets”. In: *Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, 2005, Department of Computer Science, University of Aarhus, PB-576, 211230* (2005).

[J D01] G. Juhás J. Desel. “What Is a Petri Net? Informal Answers for the Informed Reader. In Ehrig, Hartmut;” in: *nifying Petri Nets. LNCS. 2128. Springer-link.com. pp. 1–25.* (2001).

[JN03] R. Lorenz J. Desel G. Juhas and C. Neumair. “Modelling and Validation with VipTool.” In: *BPM 2003, LNCS 2678, pp. 380389, SpringerVerlag,* (2003).

[KM 08] et al. K.M. van Hee J. Keiren. “Designing case handling systems”. In: *Transactions on Petri Nets and Other Models of Concurrency I, LNCS 5100, pp. 119133, Springer, Berlin* (2008).

[M B01] et al. M. Beaudouin-Lafon W. E. Mackay. “CPN/Tools: A Tool for Editing and Simulating Coloured Petri Nets”. In: *Tools and Algorithms for the Construction and Analysis of Systems, LNCS 2031, pp. 574577, Springer-Verlag* (2001).

[M M17] et al. M. Mladoniczky G. Juhas. “Petriflow: Rapid language for modelling Petri nets with roles and data fields.” In: *Algorithms and Tools for Petri Nets, 45.* (2017).

[R B06] et al. R. Bergenthum J. Desel. “Can I Execute my Scenario in Your Net? VipTool tells you!” In: *Application and Theory of Petri Nets and Other Models of Concurrency. LNCS 4024, pp. 381390, Springer-Verlag,* (2006).

Appendix A

Supplementary Material

For newest version of this report and other supplementary material including XML scheme of the syntax of the Petriflow OCP see petriflow.org.