Process Communication in Petriflow: A Case Study

Milan Mladoniczky[1,2], Gabriel Juhás[1,2,3], and Juraj Mažári[1,2]

[1] Faculty of Electrical Engineering and Information Technology
Slovak University of Technology in Bratislava,
Ilkovičova 3, 812 19 Bratislava, Slovakia,
Home page: https://fei.stuba.sk
[2] NETGRIF, s.r.o.,
Jána Stanislava 28/A, 841 05 Bratislava, Slovakia
netgrif@netgrif.com
Home page: https://netgrif.com
[3] BIREGAL s. r. o.,
Klincova 37/B, 821 08 Bratislava, Slovakia
biregal@biregal.sk
Home page: http://www.biregal.com

Abstract. In this paper, we explain the usage of Petriflow language in a multi-process environment with inter-process communication via process events. We introduce an example that demonstrates the advantages of Petriflow language when synchronisation between two and more processes is required.

Keywords: Petri nets · workflow · Petriflow · process events · inter-process communication · Petriflow actions

1 Introduction

Petriflow[1] is modelling language for developing process-driven applications. It is based on Petri nets with an extension of reset, inhibitor, and read arcs to enable multiple concurrent readings on a transition. Petriflow can be divided into several layers. The first layer is a Petri net process model itself. Process roles are the second layer of Petriflow language. They provide access control over transitions of a process model. The third layer also called data-set of a process, consists of all data variables of a model. The fourth and the last layer are actions. Actions are small snippets of code written in Groovy programming language. Actions are a powerful tool in Petriflow language. They can define relations between data variables of a process, generate values, communicate with external services or synchronise different instances of processes.

When a Petriflow process model is deployed to an application server, to execute process, an instance of the process is created. In Petriflow, an instance of a process is also called a case. A case is a deep copy of the original process with its specific marking and data-set values. In Petriflow, each enabled transition is a task[2]. A task consists of four events: assign, cancel, delegate, finish. Each task event can be triggered, by a user or a system, to execute a specific function of a task. Petriflow provides means to react to such events via actions. It is also possible to trigger process events inside of an action and created chain of events and reaction influencing different instances of different processes.

2 Multi-process environment

The real world is very complex. It is one of many reasons why process-driven applications consist of a vast number of processes. It is important to be able to define a way of process communication. Petriflow allows to model inter-process communication with process events and actions. All events can be invoked by some system entity, like system user or another process. Also, a reaction can be defined to every process event in a form of an action. The important part of inter-process communication is the search of all entities of a process model. Petriflow provides search capabilities via library QueryDSL that is easy to use and enables to write both simple and complex queries on every entity of a process model. With Petriflow capabilities, we can model processes that can communicate with each other within an application environment.[3]

3 Process hierarchy

It is rather difficult to model a hierarchy in an observed system with original concept of Petri nets. Even more, if the observed system is very large. A net that tries to capture the hierarchy of a system often results to be large and cumbersome to work with. For modelling hierarchy and encapsulation of components of a system, Nested Petri nets[4] are usually used. Nested Petri nets model a system behaviour in different levels of detail to define relations between parts of an observed system. Nested Petri nets can become difficult to read when a modelled system has multiple levels of complexity. However, modelling hierarchy between processes can be also achieved by writing invocation and reaction to the events in processes in Petriflow language.

Let us introduce an example of this problem. An observed system, which behaviour will be synthesised into Petriflow process models, consists of three entities.

1. Volume - An abstract representation of a whole space inside the system.
2. Folder - An abstract representation of a part of the system space that is a specified part of the volume. The volume of the system can contain one or more folders. A folder also can contain one or more sub-folders.
3. File - An abstract representation of the smallest entity of the system. Every file has to be located inside a folder. A file cannot be further divided and create other system entities.

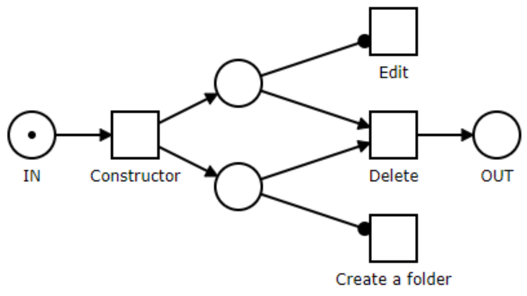

Fig. 1. The Volume process

From an analysis of the three system entities, three Petriflow models are created. Each model consists of transitions to manage the modelled system entity and its data-set.

The Volume process, in the Figure 1, contains data variables for a name of a volume instance, a name for a new folder, and an array of reference objects to all Folder process instances which are located inside the system volume.

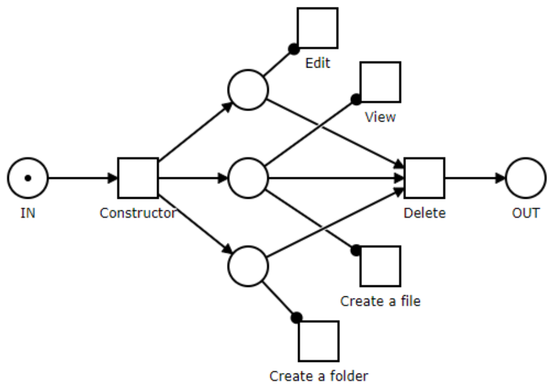

Fig. 2. The Folder process

The Figure 2 illustrates the Folder process. The process has data-set containing data variables of an array of its sub-folders and an array of its files.

The File process, in the Figure 3 is quite simple. Its data-set contains reference to its parent and raw bytes of it content.

It is clear, from the models in Figure 1, Figure 2, and Figure 3, that key transitions are `Create a folder` in the Volume process, `Constructor`, `Create a folder`, `Create a file` in the Folder process and `Constructor` in the File process. As these models are written in Petriflow language each enabled tran-

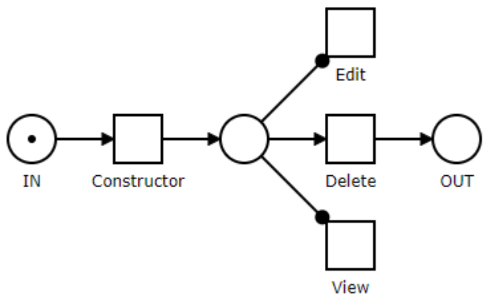

Fig. 3. The File process

sition is a Petriflow task with its process events. First, an action to the finish event on the transition `Create a folder` is defined.

Listing 1.1. An action in the Volume process to create a new folder instance

```
<event type="finish">
  <actions phase="post">
    <action>
      Case folder = createCase("Folder",folderName.value);
      change folders value {
        folders.value.add(folder.id);
        return folders.value;
      };
    <action>
  <actions>
</event>
```

When a user assigns a task `Create a folder`, fills out data about a new folder, e.g. folder's name, and then finishes the task, the action is executed. The new folder instance is constructed by calling the `createCase` function with the Folder process `id` and a new instance title stored in data variable `folderName`. The created instance is stored in the local variable `folder`. Then the `id` of the constructed instance is added to values of data variable `folders`.

This example perfectly expresses the parent-child relationship between the Volume instance and the Folder instance. Likewise, the relationship between different instances of the Folder process and between instances of the Folder and the File process can be defined.

The Second important transition in the processes is the `Constructor` transition. It is the first transition in the process and it is responsible for initialising and setting the process data of a new instance. In the example above, only a name of an instance is sent to the new instance of the Folder process. If a new file is created, it is required to set reference to the parent folder. To achieve this functionality, the required data value can be sent to the Constructor task of a newly created File instance.

Listing 1.2. An action to create a new file and pass its parent folder in the Constructor task

```
<event type="finish">
  <action phase="post">
    <action>
      Case file = createCase("File",fileName);
      change files value {
        files.value.add(file.id);
        return files.value;
      };
      Task constructor = findTask {
        it.title.eq("Constructor")
        .and(it.caseId.eq(file.id))
      };
      if(constructor){
        constructor = assignTask(constructor);
        setData(constructor,["parent":useCase.id]);
        finishTask(constructor)
      }
    </action>
  </action>
</event>
```

The action to create and set up a new file instance is called when a user finishes the task Create a file of the Folder process instance. The new file instance is created by calling the createCase function with the File process id and a name of the new file. The returned reference to the created file instance is added to the data variable of the Folder process instance file, which stores references to all files stored in the folder. The next step in the action is to send the required data to the file instance. First, the Constructor task of the new file instance is found via findTask function with QueryDSL expression parameter. The entity search is based on "Query by example" principle. As it can be seen in the example, the Listing 1.2, the keyword it in the expression is the example object of the task entity. If the Constructor task is returned, assign it to the currently logged user. The parent folder reference is set by function setData where the first parameter is the Constructor task and the second parameter is a map of data variables. The key of the map is a data variable id of the File process and the value of the map is a desired value of the data variable. In the example above, the Listing 1.2, the data variable with id parent is set to reference the current Folder process instance. At last, the Constructor task is finished.

4 Conclusion

Inter-process communication modelled in Petriflow language can be applied to the countless applications. As the example illustrated in this paper, it can be used to express hierarchy between instances of different processes. It can be also used to separate often repeated parts of a process as a standalone process model and then referenced from the original process. Even large and complex processes can be modelled with communication via Petriflow process events with ease and preserved readability of Petri nets.

References

1. Mladoniczky, M., Juhás, G., Mažari, J., Gažo, T. and Makáň, M.: Petriflow: Rapid language for modelling Petri nets with roles and data fields. Proceedings of the Workshop Algorithms and Tools for Petri nets 2017, October 19-20, 2017, Technical University of Denmark, Kgs. Lyngby, Denmark, (2017)
2. Riesz, M., Seckár, M., Juhás, G.: PetriFlow: A Petri Net Based Framework for Modelling and Control of Workflow Processes. In ACSD/Petri Nets Workshops (pp. 191-205). (2010)
3. Mažari, J., Juhás, G., Mladoniczky, M.: Petriflow in Actions:Events Call Actions Call Events. Proceedings of the Workshop Algorithms and Tools for Petri nets 2018, October 11-12, 2018, University of Augsburg, Germany, (2018)
4. Irina A. Lomazova, Philippe Schnoebelen: Some decidability results for nested Petri nets. Springer LNCS 1755, 208-220 (2000)
5. Van der Aalst, W. M.: The application of Petri nets to workflow management. Journal of circuits, systems, and computers, 8.01, 21-66 (1998)