Petriflow in Actions: Events Call Actions Call Events

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

[1] Faculty of Electrical Engineering and Information Technology
Slovak University of Technology in Bratislava,
Ilkovičova 3, 812 19 Bratislava, Slovakia,
[2] NETGRIF, s.r.o.,
Jána Stanislava 28/A, 841 05 Bratislava, Slovakia
netgrif@netgrif.com
Home page: http://www.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 present how to model complex business processes and synchronisation between their instances using Petriflow language which extends Petri Nets with other components. Small snippets of code called Actions are introduced to show the capabilities of inter-process communication. Examples of searching, constructing new instances, executing tasks, and data manipulation are provided.

Keywords: Petriflow, Business process modelling, Inter-process communication

1 Petriflow

Petriflow language is an XML based extension of Petri nets[1]. As the underlying model, we use place/transition nets enriched by reset arcs, inhibitor arcs and read arcs. The read arcs appear quite necessary in order to model an unbounded number of concurrent reading of data in a case. To meet modern business modelling requirements other layers were brought to the language on top of Petri nets. Roles are the first layer to extend Petri nets. Roles layer defines who can fire transitions to which they are bound. Data variables were added as the second layer on top of modelled processes. Data variables represent all properties of an instance of a process during its life-cycle. Data variables are bound to transitions by dataRef tag in the underlying XML creating a dataset of the transition. To have more control over process instance data, data field actions were added to Petriflow. Actions can define relations or dependencies between data fields in the model of a process or generate values based on a process instance state. All extensions and layers create the right tool for modelling complex, yet simple to understand models of any process that comes to mind.

Each transition represents a task [2] that can be executed when the transition is enabled. The task life-cycle can be modelled with a Petri net in figure 1. Each task consists of four basic events:

1. Assign - triggered when an actor starts the task execution,
2. Delegate - triggered when an actor assigns the task to another actor,
3. Cancel - triggered when an actor stops the task execution,
4. Finish - triggered when an actor finishes the task execution.

Process roles restrict which event can be triggered by which actor. Data fields can be edited only if the task is active and only by the actor executing the task.

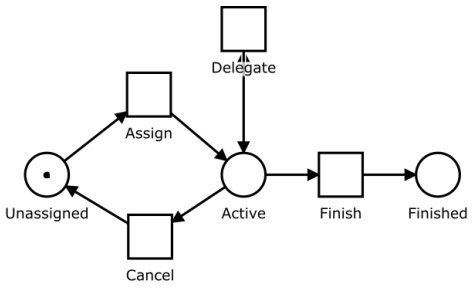

Fig. 1. Task representation net

2 Petriflow universe

Petriflow universe consists of a set of process models and their instances - cases. Each case is a deep copy of the original process model with its own set of data. This concept is similar to relational databases.

Process model represents an entity. In analogy to a relational database, Petriflow model represents a table. It defines a set of data variables one can work with, their data types and validations which is an analogy to table columns. Table rows represent instances of given entity. In Petriflow cases are instances of the original model. Set of tables forms a database, in our universe it is a process driven application working with multiple Petriflow models.

In relational databases, foreign keys are used to create relationships between tables. Since Petriflow is focused on processes we use tasks and events to transfer data between cases and change the state (marking) of a case.

3 Inter-process communication

Inter-process communication can be used to model complex real-life processes by creating an interface between distinct Petriflow models. This interface consists

of models transitions and its events, which can be triggered by an authorised actor. This means that one case can create a new instance of Petriflow model, read and write tasks data, and trigger events on tasks. Each task event (assign, delegate, cancel, finish) can be triggered by an application user or from another process using Actions.

Actions are small pieces of Groovy code, that can be executed at different places in our model. They can be used to change data variables values and behaviour, change case attributes such as title or icon. Customer specific actions can be defined in the application and used in the model as well.

Actions can be triggered by following events:

- assign task,
- delegate task,
- cancel task,
- finish task,
- read value of data field,
- set a new value of data field.

A very important feature of Petriflow is that it allows specifying if the action should be triggered before the event or after. Some actions need to be executed before the event due to change in the marking of the net or if a failure in execution of the action should prevent the event.

Imagine that we have a custom function, one updating document record in a DMS via a web service. A task will update document status on its finish event. It should not be possible to finish this task if the update fails. Other action needs to be executed after the finish event. For example, a task called `aTask` should be assigned to a user immediately after the finish event of the current task produces tokens. Both actions can be assigned to the same event.

Listing 1.1. Example of case search

```
<event type="finish">
  <actions phase="pre">
    <action>
      updateDocumentRecord()
    </action>
  </actions>
  <actions phase="post">
    <action>
      assignTask(aTask, user)
    </action>
  </actions>
</event>
```

As it was already illustrated above, actions can not only be triggered by these events, but actions can trigger events on other tasks (as an event `assignTask` assigned `aTask` to a user inside of an action). Moreover, actions can be used to search for an entity (case, transition, task etc.), create a new case and work with data of a different case.

3.1 Finding objects

Searching for information is a fundamental part of every enterprise application. In terms of inter-process communication, search action is used to find a specific instance to read or change its data values, or to find a task that should be executed in order to continue.

Actions use QueryDSL language to formulate search predicates. Search predicates use an example search object called `it`. It supports complex querying using logical operators and dynamic expressions. Every entity can be searched by any of its properties. This allows to search for a case with a specific data variable value:

Listing 1.2. Example of case search

```
List<Case> cases = findCases( {
    it.dataSet.get("email").eq("mazari@netgrif.com")
} )
Case aCase = findCase( {
    it.author.id.eq(loggedUser().id)
} )
```

For each entity, there are two find actions. One that will return a list of all entities that match the given predicate. The other will always return only one entity, even if multiple entities match the predicate. In that case, the first entity is returned.

In the example above, `it.dataSet` returns the set of all data variables of example case `it`. `get("email")` returns the data variable with id `"email"` of that dataset. The rest of the predicate compares the value of this data variable `"email"` with a string `"mazari@netgrif.com"`.

3.2 Creating new instances

Creating new cases is the basic function in each process-driven application. To create a new instance we only need a reference or the Id of the net. All the other parameters such as case title, colour and author are optional and default values will be used.

Listing 1.3. Example of case constructor

```
Case aCase = createCase("insurance", "My insurance")
```

In this example a new case will be created by calling the `createCase` action with Petriflow net Id and case title.

3.3 Triggering task event

As mentioned before, each of the four basic task events can be triggered by an action. These actions take a transition Id or task reference as the first parameter. The second parameter is the actor whose identity will be used to trigger the

event. If the parameter is omitted logged user will be used instead. Imagine that we have a net with data variable `email` and another net called `"some_net"` with a transition `"first_task"`.

Listing 1.4. Triggering task event

```
if (email.value != null) {
  def user = findUser(it.email.eq(email.value))
  def nC = createCase(identifier:"some_net",author:user)
  assignTask("first_task", nC, user)
}
```

In the example above, `email` refers to a data variable of the current case with id `"email"`. The user whose email equals to the value of the variable `email` is stored to the local variable `user`. Then a new case of `"some_net"` is constructed by the found user stored in the local variable `user`. Finally calling `assignTask` function will assign the found user to the task with id `"first_task"` of the constructed case of `"some_net"`.

3.4 Reading and writing data

Data variables in Petriflow are bound to transitions and can be accessed by reading and writing using these transitions. Since only authorised actors can execute transitions this allows you to restrict access to data values of each model using process roles. In addition, data behaviour can be defined that specifies if the value is visible, optional, required or hidden. This behaviour not only restricts the read and write operations but also restricts triggering of the finish event. If the required data variable does not have a proper value task cannot be finished.

To read the tasks dataset we need to specify the task data of which should be read. This can be done by providing:

1. reference of the task itself by calling `findTask` which returns the task object,
2. reference of the tasks transition (works only on tasks in the current case),
3. tasks id and case.

Function `getData` will use the third method of specifying the task. It will return a map where the key is the data variable Id and the value is the data variable. Returned data variable can be used to access the value of the data variable and other properties such as title, placeholder, behaviour attributes specifying whether it is required, editable, hidden, etc.

Imagine that we have a net with transition with id `"view_limit"`, with data variable with id `"actual_limit"` such that data variable `"actual_limit"` belongs to the dataset of transition `"view_limit"`.

Listing 1.5. Reading data values

```
def usecase = findCase({ it.title.eq("Limits") })
def data = getData("view_limit", usecase)
change actual_limit value {
    data["remote_limit"].value
}
```

In the example above, first case with title `"Limits"` is returned to the local variable `usecase`. Then the dataset of transition `"view_limit"` is returned to the local variable `data` in form of a map described above. Then value of the data variable `"actual_limit"` of the current case is set to the same value as the value of data variable `"remote_limit"` in first case with title `"Limits"` which is stored in the local variable `usecase`. Writing data works similarly to reading, the first parameter specifies a task, the second parameter is a map of new values with data variable Id as the key.

Listing 1.6. Writing data values

```
def aCase = findCase({ it.title.eq("Limits") })
setData("edit_limit", usecase, [ "new_limit": 10000 ])
```

The example above set the value of the data variable `"new_limit"` in the dataset of the transition `"edit_limit"` of the first case with title `"Limits"` stored in local variable `aCase` to the value 10000.

It is not mandatory to provide values of all data variables that are bound to the task. In some scenarios, you want to set new values in steps, especially if the values are dependent on each other. For example, there can be one data field where a user can enter a country name and a second data field where he has to select a city of the chosen country. In that case, you would need to first call `setData` to select the country, then call `getData` to read the list of cities and finally select one city by calling `setData` again.

4 Conclusion

Inter-process communication is the future of Petriflow. This concept has proven to be simple enough for customers to understand it and also simple to model at the same time. Multiple applications using its modelling capabilities were developed and deployed to production.

Inter-process communication opens new possibilities of modelling real-life scenarios. Hotel reservations, online shopping, accounting information system, these are just examples of applications that can be now developed using Petriflow. Whole user management could be replaced with a right Petriflow model allowing to further customise the application.

References

1. Mladoniczky, M., Juhás, G., Mažári, J., Gažo, T., Makáň, M.: Petriflow: Rapid language for modelling Petri nets with roles and data fields. Algorithms and Tools for Petri Nets, 45. (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. Van der Aalst, W. M.: The application of Petri nets to workflow management. Journal of circuits, systems, and computers, 8.01, 21-66. (1998)