A Process Scripting and Execution Environment

Maxim Vidgof[1], Philipp Waibel[1], Jan Mendling[1], Martin Schimak[2], Alexander Seik[3], and Peter Queteschiner[3]

[1] Institute for Information Business, WU Wien, Austria
{firstname.lastname}@wu.ac.at
[2] Plexiti, Vienna, Austria martin.schimak@plexiti.com
[3] Phactum, Vienna, Austria {firstname.lastname}@phactum.at

Abstract. Many companies rely on business process management systems for their digital transformation initiatives. Beyond visual modeling languages, there is a need for an executable modeling language with data processing support and powerful abstractions for development. To overcome this gap, we developed a process scripting language called Factscript. In this paper, we present the Factscript language environment and showcase its application.

Keywords: Process Modelling · DSL · Kotlin · Factscript

1 Introduction

Business Process Management Systems (BPMS) [3] support the implementation of specific processes. Several BPMS solutions and languages have been proposed in the past, e.g., BPMN [1], BPEL [4], Netflix Conductor or Uber Cadence, each with strengths and weaknesses. While BPMN supports important control flow patterns, it does not support proper data processing on the visual level. On the other hand, languages like Netflix Conductor are process programming languages with data processing support but are missing important abstractions.

In order to overcome the mutual weaknesses of the different languages, we developed a process scripting language, called Factscript [6] and supportive tools. Factscript is an executable, code-efficient, domain-specific language (DSL) that builds on the strengths of different process languages.

In this paper, we present the language environment of Factscript. The language environment supports the user during the design of the processes in Factscript, deploys the processes to an external execution engine, and executes the processes. Furthermore, we present a real-world application scenario, composed of two interconnected processes, for Factscript and the language environment.

2 Language Environment

In the following, we discuss the features of our scripting language and introduce the language environment.

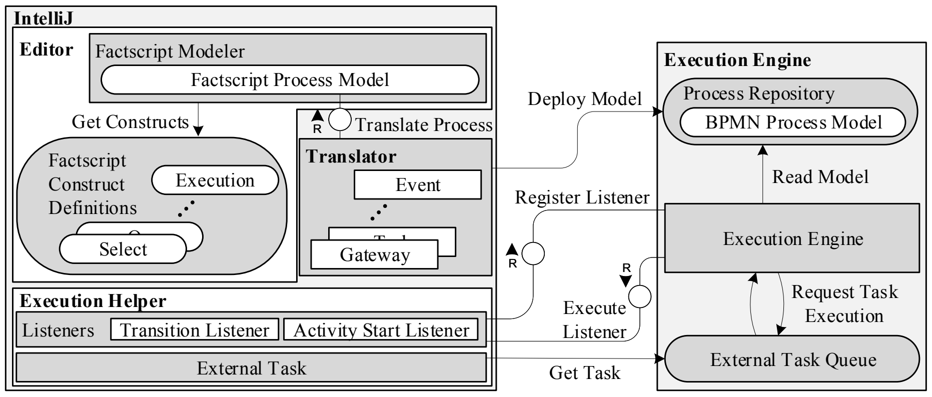


Fig. 1: Implementation Architecture

For the development of the Factscript, we identified four requirements [6]: (R1) efficient syntax, (R2) extensive control flow support, (R3) support of data processing, and (R4) support of event processing. We followed the seven phases of DSL development [2,5], which helped us to generate an efficient syntax (R1). For (R2) we analyzed process modeling languages and their control flow support. For extensive control flow support [8], we refer to BPMN. For data processing (R3), we identify type safe variables and their manipulation directly in the process definition as important. For (R4) we opted for BPEL as a basis with its different message interaction possibilities [7]. Factscript, therefore, provides different constructs that permit a similar message interactions as BPEL does.

The language environment is composed of four top-level entities: *editor*, *translator*, *execution engine*, and *execution helper*. The editor entity is used for scripting a process in Factscript by using the Factscript constructs and syntax. Depending on the editor, the scripting is supported by auto-completion and syntax checking. After the process is defined, the translator entity is used to transform the Factscript process into a format that can be interpreted by the execution engine. Eventually, the execution engine executes the process. Dependent on the functionalities of the used execution engine, different execution helper functions are required. The particular implementation of these entities is independent of each other and can be based on different tools.

At the current development state, the implemented language environment for Factscript uses IntelliJ IDEA for the editor, translator, and execution helper entities. These entities are implemented using Kotlin. Camunda BPM is used as execution engine . Figure 1 depicts the architecture of the current implementation. The language environment, as well as the Factscript construct definitions and example processes, can be accessed online[4] and in [6].

[4]`https://github.com/factdriven` (commit tag ER-2020-Demo).

3 Application Scenario

In the following, we present a real-world application scenario, supplementary to the one presented in the screencast of the Factscript environment. The screencast can be accessed at `https://youtu.be/WBs-nC1S8OI`.

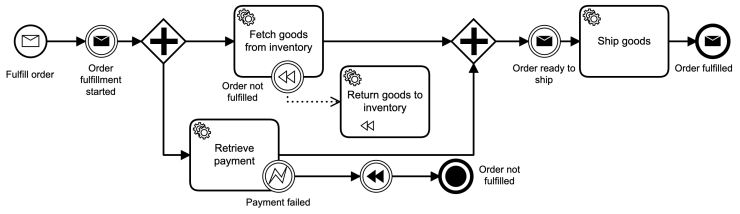
The application scenario is composed of two processes: The first one is an order process, and the second one a payment process that is used by the order process. Listing 1.1 shows the order process and Listing 1.2 the payment process defined in Factscript. A description of the semantic and syntax of Factscript can be found in [6]. Figures 2a and 2b present the processes in BPMN 2.0.

Listing 1.1: Order Process in Factscript

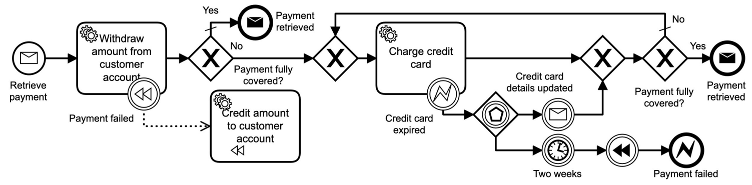
```
on command FulfillOrder::class emit {
  success event OrderFulfilled::class
  failure event OrderNotFulfilled::class
}
emit event { OrderFulfillmentStarted(orderId, accountId, total) }
execute all {
  execute command {
    FetchGoodsFromInventory (orderId)
  } but {
    on failure OrderNotFulfilled::class
    execute command { ReturnGoodsToInventory(orderId) }
  }
} and {
  execute command {
    RetrievePayment(orderId, accountId, total)
  } but {
    on failure PaymentFailed::class
    emit failure event { OrderNotFulfilled(orderId) }
  }
}
emit event { OrderReadyToShip(orderId) }
execute command { ShipGoods(orderId) }
emit success event { OrderFulfilled(orderId) }
```

Listing 1.2: Payment Process in Factscript

```
on command RetrievePayment::class emit {
  success event PaymentRetrieved::class
  failure event PaymentFailed::class
}
execute command {
  WithdrawAmountFromCustomerAccount(customer = accountId, withdraw = total)
} but {
  on failure PaymentFailed::class
  execute command {
    CreditAmountToCustomerAccount(customer = accountId, credit = covered)
  }
}
select ("Payment fully covered?") either {
  given ("No") condition { covered < total }
  repeat {
    execute command {
      ChargeCreditCard(orderId, total - covered)
    } but {
      on failure CreditCardExpired::class
      await first {
        on event CreditCardDetailsUpdated::class having "accountId" match {
            accountId }
      } or {
        on time duration ("Two weeks") { "PT5M" }
24        emit failure event { PaymentFailed(orderId) }
25      }
26    }
27    until ("Payment fully covered?") condition { covered == total }
28    }
29 } or {
30   otherwise ("Yes")
31   emit success event { PaymentRetrieved(orderId) }
32 }
```



(a) Order Process in BPMN 2.0



(b) Payment Process in BPMN 2.0

Fig. 2: Order and Payment Process in BPMN 2.0

The order process (Listing 1.1) performs the following steps: (1) The execution is triggered by a *fulfill order* event (lines 1-4). (2) At the beginning of the process execution, an *order fulfillment started* event is emitted (line 5). (3) Subsequently, the ordered goods are fetched from the inventory, and, in parallel, the payment is retrieved via the separate payment process (lines 6-20). If the payment fails, a compensation is triggered that returns the goods to the inventory (lines 16-19). (4) After the goods are available and the payment is fully covered, an *order ready to ship* event is triggered, and the goods are shipped (lines 21-22). (5) The process ends with a *Order fulfilled* event (line 23).

The payment process (Listing 1.2) performs the following steps: (1) The execution is triggered by a *retrieve payment* event, which is emitted by the *Retrieve payment* activity of the order process (lines 1-4). (2) After the execution is triggered, the money is withdrawn from the customer's account (lines 5-12). (3) If not all of the money can be withdrawn (line 13), the customer's credit card is charged (lines 14-29). (4) If the credit card of the customer is expired (line 19), the customer has two weeks to update the credit card details. After the two weeks, the payment process fails, and the already withdrawn money is credited to the customer's account (lines 18-26). (5) After charging the credit card, it is checked if the payment is fully covered (lines 13). If not, the credit card is charged again. (6) If the payment is fully covered (lines 29-32), the process signalizes its end with a *payment retrieved* event.

This application scenario shows a part of the rich spectrum of constructs that the Factscript language provides and the possibility to create real-world scenarios with them.

4 Conclusion

In this paper, we presented the language environment for Factscript. The language environment provides the means to script processes by using the Factscript constructs, then deploys the processes to an external execution engine that is then used to execute the process. At the current level of maturity, the Factscript already fulfills all four requirements discussed in Section 2. Furthermore, the language environment, with Camunda BPM as an execution engine, fully supports all Factscript constructs. Since the Factscript and execution engine are independent of each other, the execution engine can easily be exchanged. As future work, we integrate Factscript with Amazon AWS Step Functions as a process execution engine.

Acknowledgement The concept and design of the Factscript as well as its implementation in Kotlin was done by Martin Schimak, plexiti GmbH.

This research is partially funded by FFG (ref. no. 876062).

References

1. Business process model and notation (bpmn) version 2.0. Tech. rep. (2011)
2. Ceh, I., Crepinsek, M., Kosar, T., Mernik, M.: Ontology driven development of domain-specific languages. Comp. Sci. Inf. Syst. **8**(2), 317–342 (2011)
3. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management, Second Edition. Springer (2018)
4. Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., et al.: Web services business process execution language version 2.0. Tech. rep. (2007)
5. Mernik, M., Hrncic, D., Bryant, B.R., Javed, F.: Applications of Grammatical Inference in Software Engineering: Domain Specific Language Development. In: Scientific Applications of Language Methods, vol. 2, pp. 421–457 (2010)
6. Vidgof, M., Waibel, P., Mendling, J., Schimak, M., Seik, A., Queteschiner, P.: A Code-efficient Process Scripting Language. In: ER Conference Proceedings) (2020)
7. Weidlich, M., Decker, G., Großkopf, A., Weske, M.: BPEL to BPMN: the myth of a straight-forward mapping. In: On the Move to Meaningful Internet Systems: OTM. Lecture Notes in Comp. Sci., vol. 5331, pp. 265–282. Springer (2008)
8. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H., Russell, N.: On the suitability of bpmn for business process modelling. In: Int. Conf. on business process management. pp. 161–176. Springer (2006)