NextFlow: Business Process Meets Mapping Frameworks

Rogel Garcia, Marco Tulio Valente

Department of Computer Science, UFMG, Brazil

{rogelgarcia,mtov}@dcc.ufmg.br

1 Introduction

Information systems (IS) increasingly rely on business process models, notations, and engines to represent and execute complex business rules [2, 5]. However, the integration between IS and current Business Process Management Systems (BPMS) is usually based on low-level programming interfaces that expose many accidental complexities typical from business process implementations [3].

To tackle this problem, we report in this article the design and implementation of a Java-based mapping framework—called NextFlow—that provides a high-level API for communication with BPMS. Our inspiration for designing NextFlow were the Object-Relational Mapping (ORM) frameworks that are widely used to shield IS from low-level data structures provided by Relational Database Management Systems (RDBMS). Although RDBMS and BPMS have different purposes, we argue that mapping frameworks can bring to BPMS clients similar benefits than ORM provides to systems using RDBMS. Particularly, by relying on NextFlow, IS can be oblivious to low-level implementation details of current BPMS. Moreover, it is possible to change the underlying BPMS engine without impacting other IS components.

2 NextFlow in a Nutshell

In NextFlow, a business process is mapped to object-oriented elements. Figure 1 presents the main steps that must be followed to use the proposed framework. The first step is executed at development time. In this step, object-oriented artifacts—such as classes, interfaces and methods—are created by IS developers to represent and to interact with business processes. At runtime, the IS access the mapped object-oriented elements (step 2). Particularly, method calls are intercepted by NextFlow and translated to specific BPMS commands (step 3), which are executed by the underlying BPMS engine (step 4).

To define a mapping between processes and objects, NextFlow relies on an abstract model that represents the central elements in a business process. For the design representation, we assume that a business process is a directed graph. The nodes in this graph and their relationships constitute

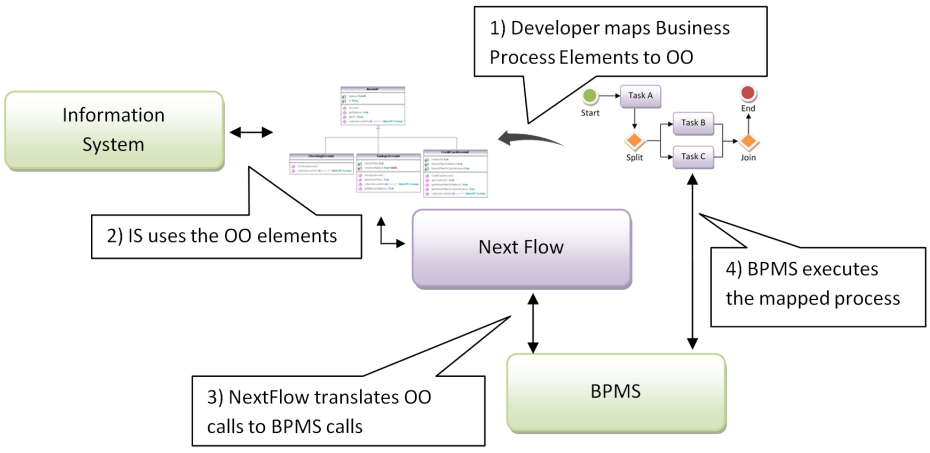

Figure 1: Integrating IS and BPMS with NextFlow

a Process Definition. A node in a process definition is an Activity Definition, which can be of the following types: *start*, *end*, *split*, *join*, *task*, and *external task*. For the execution representation, we assume that a running process constitutes a Process Instance. Moreover, the runtime counterpart of an activity definition is an Activity Instance. Figure 2 shows a graphical example of a process definition with the elements proposed by the NextFlow Model.

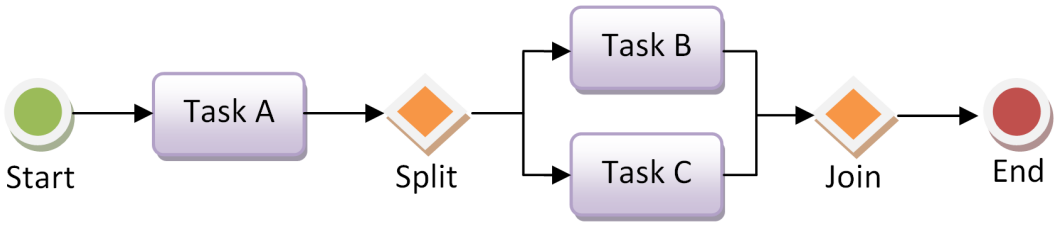

Figure 2: Example of Process Definition

3 Mapping Rules

To present the mapping rules proposed by NextFlow, a simple banking loan process is used as example. This process, as showed in Figure 3, has a single external task, called *Approve Transaction*. Despite its minimal size, this process is sufficient to illustrate our mapping rules.

Figure 3: Mininmal Loan Process Definition

3.1 Mapping Process and External Tasks

The basic functionality that IS requires from BPMS is the execution of external tasks. In a typical scenario, the user fills a form in the IS and clicks submit. Then, the IS delegates to the BPMS the execution of and external task with the parameters provided by the user. Therefore, the first element that needs to be mapped is the process itself. As illustrated by the following code, a business process is mapped to a Java interface—called *process interface*—that establishes a contract between the IS and the BPMS.

```
1 @Process("loanprocess")
2 interface LoanProcess {
3   void approveTransaction();
4 }
```

The `@Process` annotation indicates the ID of the process. Moreover, external tasks are mapped to methods in this interface. By calling these methods, the IS can trigger the execution of the associated task in the BPMS. The concrete class implementing a process interface is provided at runtime by NextFlow.

3.2 Mapping Process Data

Typically, a business process manipulates some global data, stored in the BPMS [1]. In NextFlow, this data is represented by a set of key-value pairs, which we refer as the *process dataset*. In order to represent this dataset, a *data class* must be created. For example, in our loan process a possible process data is the client identification. The following data class must be defined to expose this information to the IS:

```
1 class LoanData {
2   String client;
3
4   Client getClient() { return client; }
5   void setClient(Client client) { this.client= client; }
6 }
```

NextFlow keeps the values of the attributes in a data class synchronized with the respective global values in the BPMS. For example, the attribute `client` (line 2) is associated to the key-value pair in the business process whose key is `"client"`. Therefore, the IS can read or write to the process dataset by accessing this attribute, typically by means of conventional getters (line 4) and setters (line 5).

To retrieve an object of a data class, a `get` method must be created in the process interface. As illustrated by the following code, the `LoanProcess` interface can include a `getLoanData()` method (line 4), for accessing the respective data class object.

```
1 @Process("loanprocess")
2 interface LoanProcess {
3   void approveTransaction();
```

```
4     LoanData getLoanData();
5   }
```

Besides a global process dataset, NextFlow assumes the existence of two local atasets in each external task, to store parameters and results, respectively. Attributes from external tasks parameter types are mapped to key-value pairs in the task parameters dataset. Moreover, because methods in Java cannot have multiple return values, a class must be created to represent possible external task results. As in the previous cases, the attributes in this class are mapped to key-value pairs in the task results dataset. To illustrate, in the following code we changed the signature of the `approveTransaction` method. The new signature includes a parameter of the type `Number` and a return value of type `TransactionInfo` (line 3), which is a class we created to store the values in the task results dataset (lines 7-10).

```
1  @Process("loanprocess")
2  interface LoanProcess {
3    LoanData getLoanData();
4    TransactionInfo approveTransaction(Number money);
5  }
6  class TransactionInfo {
7    Number transactionID;
8    Date transactionDate;
9  }
```

3.3 Callbacks

When tasks are executed by an underlying BPMS engine some extra computation might be required. Usually, it is possible to implement extra task semantics using the BPMS GUI, for instance by writing code in property boxes. However, this approach is not recommended because BPMS cannot compete with contemporary IDEs, which provide features like code completion, syntax highlight, automatic refactoring, syntax and type checking etc. A preferred strategy is to require the BPMS to call back services implemented by the IS. For this purpose, NextFlow provides support to *callback classes*, whose methods are automatically called by NextFlow when tasks with the same name (not necessarily external tasks) are executed by the BPMS. The following code illustrates the definition of a callback class for our loan process example.

```
1  @Process("loanprocess")
2  class LoanProcessCallback {
3    TransactionInfo approveTransaction(Number value) {
4      // extra semantics required to approve transactions
5      TransactionInfo info = ...;
6      return info;
7    }
8  }
```

4 Architecture

As illustrated in Figure 4, NextFlow has an internal architecture with two layers. The first layer, called *Workflow Connectivity* (WFC), connects NextFlow to the concrete business process model of a given BPMS implementation. The second layer, called *Object-Workflow Mapping* (OWM), implements the mapping rules described in the previous section. For readers familiar with Java, the WFC layer represents to business processes what JDBC is for relational databases, and the OWM layer is analogous to an ORM framework, like Hibernate (`www.hibernate.org`).

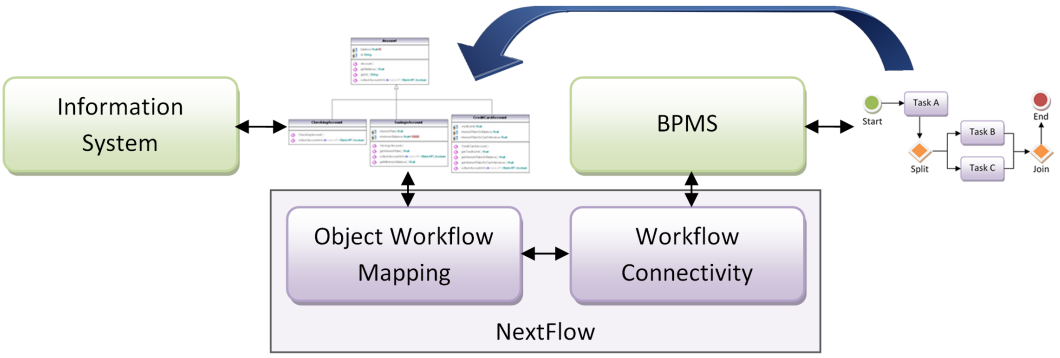

Figure 4: NextFlow architecture

The WFC layer provides abstract interfaces exposing elements from the NextFlow Model, including interfaces like `ProcessDefinition`, `ProcessInstance`, `ActivityDefinition`, etc. The concrete implementation of this layer is supported by *drivers* that implement these interfaces for a particular BPMS.

Regarding the OWM layer, a central component is the `WorkflowObjectFactory` class, which acts a communication port with the client IS. Particularly, this class provides methods for starting new process instances and to retrieve running processes. The following code shows how this class can be used to start a new instance of our loan process and to call an `approveTransaction` task.

```
1 Configuration configuration= new Configuration("jwfc:jBPM:loanprocess.jbpm");
2 configuration.addCallbackClass(LoanProcessCallback.class);
3 WorkflowObjectFactory factory= configuration.createFactory();
4 LoanProcess loanProcess= factory.start(LoanProcess.class);
5 TransactionInfo result= loanProcess.approveTransaction(1000);
```

First, a configuration is created with a URL that defines the underlying BPMS engine—in this case, jBPM—and a BPMS specific file with the definition of the business process in a given modeling language (line 1). Next, a callback class is bound to this configuration (line 2). Using this configuration as target, an `WorkflowObjectFactory` is created (line 3). Using this factory, we start a business process instance (line 4) and finally we call an external task of the started process.

5 Comparison with jBPM

In the architecture considered by NextFlow, the BPMS is an external component that manages the execution of business process workflows (like a DBMS is the external component used by most enterprise software architectures to persist data). On the other hand, current BPMS, like jBPM, promote a different architecture, where the BPMS acts as a framework that keeps control of the main application flow. In fact, we argue that this architecture is an important obstacle for the widespread adoption of BPMS, because it hampers the reuse of popular libraries and frameworks targeting other concerns also important in large and complex IS, like presentation, transactions, persistence, logging, etc.

Anyway, we describe an alternative provided by jBPM to reuse its business process engine as an external component. In this alternative, the IS must implement methods for each external task handled by the BPMS. A possible example for our loan process is presented next:

```
1 TransactionInfo executeApproveTransaction(StatefulKnowledgeSession kSession,
2                                           WorkItemNodeInstance wi, int value){
3   TransactionInfo ti= IS.approveTransaction(value);
4   Map<String, Object>() results= new HashMap<String, Object>();
5   results.put("transactionInfo", ti);
6   kSession.getWorkItemManager().completeWorkItem(wi.getNodeId(), results);
7   return ti;
8 }
```

First, this method callbacks the method from the IS that performs the extra computation required by the task (line 2). Next, it triggers the execution of the external task by the jBPM engine (line 6), by calling the method `completeWorkItem` provided by jBPM (line 6) passing two arguments: the work item that denotes the external task to be completed and a dataset with the results returned by the callback method. We argue that the similar code in NextFlow—presented in Section 4—has two major benefits. First, it is not coupled to data types from a particular BPMS (like `StatefulKnowledgeSession`, `WorkItemNodeInstance`, etc in the presented code). Second, it provides a high-level API for accessing BPMS, that abstracts out many accidental complexities typical from current BPMS implementations (like the work item concept, which is the central abstraction manipulated by the presented code).

6 Sidebar: Related Work on Integrating Business Process with Information Systems

The Workflow Client API (WAPI) is a set of interfaces for interoperability with BPMS proposed by the WfC [6]. Particularly, WAPI specifications include two interfaces that are more close to NextFlow: the Interface 2 defines means for an IS to call services provided by BPMS implementations and the Interface 3 defines IS-based services, called *tool agents*, which are similar to the callback methods proposed by NextFlow. Therefore, WAPI aims to provide a standard API that

should be followed by BPMS implementations. However, this requirement usually represents a burden to BPMS developers. In fact, to the best of our knowledge, WAPI is not supported by any of the major BPMS implementors. On the other hand, NextFlow follows a different approach, based on mapping frameworks. Particularly, NextFlow does not force BPMS to provide a standard API but rely on drivers to access particular BPMS APIs.

Micro-Workflow is an object-oriented framework to implement business processes [4]. Following traditional framework principles, Micro-Workflow provides interfaces and components that IS-developers should implement, extend or compose to generate an application with support to BPMS services. Therefore, as in NextFlow, Micro-WorkFlow is a solution that can easily coexist with current software architectures, frameworks, and libraries. On the other hand, because the solution provides its own components and interfaces to implement a business process engine, Micro-WorkFlow represents a depart from current business process languages, models, and systems.

7 Concluding Remarks

NextFlow represents an alternative to integrate business process and information systems, by means of what we are calling an Object-Business Process Mapping (OBPM) framework. We claim that OBPM can contribute to a broader adoption of BPMS, because they do not represent a disruptive technology regarding current and widely established software architectures, frameworks, and libraries. Currently, our prototype implementation is available only for Java-based information systems. Moreover, we only provide driver support to jBPM. In the near future, we have plans to support new languages and BPMS.

Acknowledgments

This research has been supported by grants from FAPEMIG and CNPq.

References

[1] Wil Aalst and Arthur Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[2] Wil Aalst, Arthur Hofstede, and Mathias Weske. Business process management: A survey. In *1st International Conference on Business Process Management (BPM)*, pages 1–12, 2003.

[3] Michael Havey. *Essential business process modeling.* O'Reilly, 2005.

[4] Dragos Manolescu. *Micro-workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development.* PhD thesis, University of Illinois, 2001.

[5] Jan C. Recker. Opportunities and constraints: the current struggle with BPMN. *Business Process Management Journal*, 16(1):181–201, 2010.

[6] Workflow Management Coalition. Workflow Management Application Programming Interface (Interface 2 & 3) Specification, 1999.