Object–Business Process Mapping Frameworks: Abstractions, Architecture, and Implementation

Rogel Garcia and Marco Tulio Valente

Department of Computer Science, UFMG, Brazil
{rogel.garcia,mtov}@dcc.ufmg.br

***Abstract*—The integration between enterprise architectures and Business Process Management Systems (BPMS) is currently based on low-level programming interfaces that expose accidental complexities typical of process implementations. This paper describes an approach for integrating software architectures and BPMSs, based on mapping frameworks. Our inspiration are the Object-Relational Mapping (ORM) frameworks widely used to shield information systems from low-level structures exposed by relational database systems. The paper describes the central abstractions that should be provided by Object–Business Process Mapping Frameworks (OBPM). We also propose a reference architecture for implementing OBPMs and a concrete OBPM implementation, called NextFlow. We evaluated our approach by comparing two implementations of the same system, one using NextFlow and another using the native API supported by jBPM, a popular BPMS. By using NextFlow, we achieved a reduction of 30% in terms of lines of code, 35% in terms of number of classes, and 90% in terms of import statements, when implementing this system.**

***Index Terms*—Business Process; Enterprise Architectures; Mapping Frameworks; Object–Business Process Mapping Frameworks; Business Process Management Systems**

I. Introduction

Enterprise software development increasingly relies on frameworks and architectures to promote reuse and increase productivity. Particularly, enterprise architectures are usually organized in layers that confine the implementation of particular concerns. For example, normally there is a layer that responds for user interface concerns (presentation layer) and a layer responsible for persistence (data source layer) [1, 2]. The business rules are implemented by a layer called domain, which implements concerns related to the core business workflows and rules. On the other hand, with the constant pressure for optimized processes, specialized software infrastructures—generically called *Business Process Management Systems (BPMS)*—were proposed to deal with business workflows [3, 4, 5]. A BPMS supports the definition, execution, registration, and control of business processes, just like a Database Management System (DBMS) handles data services.

There are two alternatives when considering the use of BPMSs in current software systems. The first one relies on the BPMS to implement and deploy a complete application, possibly using model-based or other (semi-)automatic code generation techniques. However, using a BPMS as full-fledged software infrastructure has important limitations [6]. It is not possible to take benefit, for example, from modern and widely employed frameworks, like GUI and MVC-based application frameworks. Another problem concerns the implementation of tasks, which typically requires the definition of code in specific components of the business process definition, often using property boxes. The second alternative relies on a BPMS just to support the business workflow required by the domain layer of the information system architecture. Therefore, this second alternative requires an interface between BPMS and the remaining components of the software architecture. However, the integration between enterprise architecture components and current BPMSs suffers from important drawbacks. For example, it is harder to change the BPMS once one is established. Moreover, current APIs for accessing BPMSs expose several low-level abstractions, which can be seen as accidental complexities [7]. For example, developers have to manipulate elements like *tasks* and *nodes*, which are not part of the business semantics.

To tackle the aforementioned problems, we propose a software engineering solution for integrating enterprise architectures and BPMSs, based on mapping frameworks. Our inspiration for making this proposal are the Object-Relational Mapping (ORM) frameworks widely used to shield information systems 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 systems using BPMS the same benefits that ORM frameworks provide to systems that use databases.

We make the following contributions in this paper:

- We propose a new type of mapping framework, called Object–Business Process Mapping (OBPM) framework, to integrate BPMSs with modern software architectures (Section III). To decouple OBPMs from concrete business process notations, we propose that OBPMs should rely on an abstract business process model that includes only the minimum elements for manipulating business processes from information systems. We also define a set of mapping rules to associate the elements proposed in this abstract model to object-oriented elements. More specifically, we propose that OBPMs should provide three key abstractions: process interfaces (used by clients to trigger operations in the BPMS), data classes (for sharing data between information systems and BPMS), and callback classes (for adding external semantics to business processes).

- We propose a reference architecture for implementing OBPM frameworks (Section IV). This architecture is centered on two layers. The first layer, called *Workflow Connectivity (WFC)*, provides means to connect the abstract model to a concrete BPMS implementation. The second layer, called *Object-Workflow Mapping (OWM)*, provides an API that represents, in terms of object-oriented abstractions, the elements of a business process. Comparing with database frameworks and drivers, the WFC layer represents to business processes what JDBC is for databases, and the OWM layer is analogous to an ORM framework, like Hibernate.
- We provide a real implementation for an OBPM, called NextFlow (Section V), which follows the proposed reference architecture.
- We report an experience on using NextFlow to integrate an information system and a small but realistic business process supported by a BPMS (Section VI). In this study, we compare two possible integration scenarios: (a) using OBPM abstractions, as supported by NextFlow; (b) using the native API supported by jBPM, a well-known BPMS.

II. Background and Related Work

We organized background and related work in two subsections: business process notations, languages, and tools (Section II-A) and object-relational mapping frameworks (Section II-B).

A. Business Process Notations, Languages, and Tools

BPEL (Business Process Execution Language) is an XML-based language that assumes that business processes have a Web Service interface [8]. Therefore, BPEL guarantees interoperability between systems implemented using different technologies. However, this benefit comes at the cost of deploying every working unit as a web service. BPMN (Business Process Modeling Notation) aims to provide a standard language for workflow modeling, while being comprehensive by business participants [9]. However, BPMN does not define a reference API, and therefore BPMN engines usually provide their own proprietary APIs.

Micro-Workflow is an object-oriented framework to implement business processes [10]. 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, Micro-WorkFlow applications can coexist with current software architectures, frameworks, and libraries. On the other hand, because the framework provides its own components and interfaces to implement a business process engine, Micro-WorkFlow is not compatible with current business process languages, models, and systems.

The Workflow Client API (WAPI) is a specification from the WfMC aiming to promote the interoperability among workflow systems [11]. Considering the five interfaces defined by the WAPI standard, two are directly related to this work. The Workflow Client Applications API (Interface 2) defines a standard API for clients that need to execute BPMS operations. The Invoked Applications API (Interface 3) provides an API for extending BPMS systems, by means of *tool agents*. However, WAPI only includes a textual specification, which does not have a reference implementation in a real programming language. For this reason, BPMS developers have to implement their own concrete API. As a consequence, there is no guarantee that these concrete implementations will be compatible, at least at the syntactical level. In fact, to the best of our knowledge, WAPI is not supported by any of the major BPMS implementers.

Table I shows a comparison of the aforementioned solutions.

TABLE I
Comparison of existing Business Process (BP) languages, frameworks and APIs

	BPMN	**Micro-Workflow**	**WAPI**
Focus	BP modeling	BP implementation, using framework concepts	Interoperability between workflow systems
Pros	Widely-used	Compatibility with current architectures	API standardization effort
Cons	Engines have proprietary APIs	Incompatibility with BP languages	No concrete implementation

B. Object-Relational Mapping Frameworks

Frameworks for mapping relational databases to object-oriented elements, known as Object-Relational Mapping (ORM) frameworks, are widely used nowadays when implementing information systems in major object-oriented languages. Basically, an ORM maps database tables and columns to object-oriented classes and fields. An ORM implementation provides mechanisms to associate such elements and it controls at runtime the synchronization of data between the object-oriented system and the database. Therefore, by using ORM frameworks, it is possible to query and persist complete objects. Otherwise, developers would have to write code to query the database and read the column values, create objects for each row, and set the corresponding field values.

Listing 1 shows a method that retrieves a list of customers from a database (lines 1–5) and stores them in a list of `Customer` objects (lines 6–10), using native JBDC operations.

```
1  List<Customer> listCustomers(Connection conn) {
2    String query = "select * from customer";
3    ResultSet rs = conn.prepareStatement(query)
4                       .executeQuery();
5    List<Customer> list = new ...;
6    while(rs.next()){
7        Customer c = new Customer();
8        c.setId(rs.getInt("id"));
9        c.setName(rs.getString("name"));
10       list.add(c);
11   }
12   return list;
13 }
```

Listing 1. Retrieving a list of customers using JDBC

Listing 2 shows the same method, but using Hibernate, a popular ORM framework for Java.[1]

[1] http://www.hibernate.org

```
1  List<Customer> listCustomers(Session session) {
2    List<Customer> list = session
3                    .createQuery("from Customer")
4                    .list();
5    return list;
6  }
```

Listing 2. Retrieving a list of customers using Hibernate

By using Hibernate, it is only necessary to specify the class mapped to the database (line 2). The association between the values retrieved from the database and objects is managed by the ORM framework. In this way, developers do not need to handle columns or tables, which can be seen as accidental complexities inherent to database systems. In fact, Hibernate relies on JDBC for accessing the database, but this low-level API is not directly exposed to ORM clients. In this paper, we propose a mapping framework inspired by ORMs, but for integration between information systems and BPMSs.

III. Object-Business Process Mapping Frameworks

An Object–Business Process Mapping Framework (OBPM)—whose proposal is the central contribution of this paper—provides object-oriented elements to interact with business process management systems. Figure 1 presents the main steps that must be followed when using OBPMs in the implementation of information systems. As usual in mapping frameworks, object-oriented artifacts—such as classes, interfaces, and methods—are initially created and mapped to business processes elements (step 1). At runtime, the information system accesses the mapped elements (step 2). More specifically, each call on methods of these elements is intercepted by the OBPM and translated to a specific BPMS operation (step 3). Finally, the BPMS executes the corresponding elements in the business process (step 4).

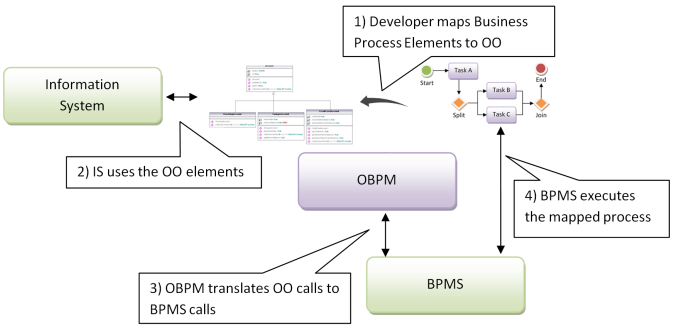

Fig. 1. Implementing information systems using Object-Business Process Mapping Frameworks

A. Abstract Business Process Model

Different business process modeling notations are supported by current BPMSs [12]. Therefore, an OBPM should not use any of such notations, because it would couple the system to that specific model. Instead, we claim that OBPMs should rely on an abstract business process model for communication with the concrete model provided by an underlying BPMS. This abstract model should define only the minimum elements for manipulating business processes as object-oriented abstractions. For example, BPMN defines several types of tasks, including normal tasks, loop tasks, multiple instance tasks, and compensation tasks [9]. However, the differences in the behavior of these tasks are not important for their mapping to object-oriented elements. Instead, we can rely on a generic *task* element to represent all possible kinds of tasks.

The proposed *Abstract Business Process Model* (ABPM) defines abstractions concerning both the design phase, i.e., how the business process components are statically organized to accomplish a desired objective and also the execution phase, i.e., how the processes are executed by a given BPMS. In the *design* phase, the business process is modeled as a directed graph. The organization of the nodes and their relationship is a *Process Definition*. Nodes in a process definition are called an *Activity Definition* and they can be of the following types: *start*, *end*, *split*, *join*, *task*, and *external task*. The *start* and *end* types denote the start and the end of the process, respectively. *Splits* are used for process parallelization, i.e., they divide the flow in multiple paths. *Joins* are used for synchronization purposes, i.e., multiple flow paths are joined in a single path. *Tasks* and *external tasks* define the work to be done. Figure 2 presents an example of a process definition in ABPM.

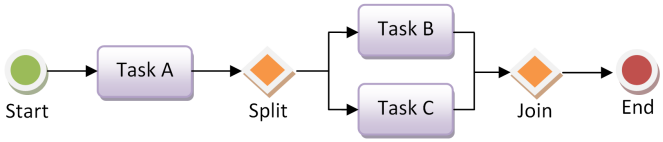

Fig. 2. Example of process definition

Concerning the *execution* phase, a running process is called a *Process Instance* and the runtime counterpart of an activity definition is an *Activity Instance*. An *automatic task* (or just a *task*, for the sake of simplicity) is automatically executed when the flow reaches the activity. On the other hand, an *external task* is only executed when triggered by an external system, possibly by the information system.

Rationale: The proposed abstract model is inspired by the model assumed by the WAPI specification [11]. As an example, a *Process Instance* in our abstract model has the same semantics as the concept with the same name in WAPI. Moreover, the task types proposed by ABPM are inspired by a work of Aalst [13]. In this work, the author states that there are four types of tasks: automatic, event, user, and timed. In ABPM, automatic and user tasks are mapped to tasks and external tasks, respectively. An event task is triggered by an external system, therefore denoting an external task in our model. A timed task is executed when a timer reaches a given timeout. Therefore, if this timer is internal to the BPMS, it is an ABPM task. If the timer is external, it is an external task. In summary, the two types of tasks defined by ABPM are generic enough to represent the variety of tasks provided by current business process languages and tools. *Join* and *split*

definitions are also present in other works [14]. According to Aalst [13], there are two types of join nodes: AND-JOIN and XOR-JOIN. In ABPM, these two types of nodes are merged in a generic *join* element. It is the underlying BPMS engine that resolves if the node can be executed and the actual semantics of its execution. The same happens with *split* elements.

Formal Definition: The ABPM is formally defined as a pair (A, C) where A is a set of nodes, called Activity Definitions, and C is a set of directed edges (or connectors). An Activity Definition is a tuple $(Type, Name)$, where $Type \in \{Start, End, Split, Join, AutomaticTask, ExternalTask\}$. The following constraints also apply to ABPMs:

1) There is only one Start Activity, which can have only one outgoing connector.
2) There is only one End Activity, which can have only one incoming connector.
3) Split Activities have only one incoming connector, but can have multiple outgoing connectors.
4) Join Activities have multiple incoming connectors, but must have only one outgoing connector.
5) Automatic and External Tasks must have only one incoming and only one outgoing connector.

B. Mapping Rules and Abstractions

As illustrated in Figure 1, the first step when using an OBPM is to map the elements defined in a business process model—more specifically, the elements considered by the ABPM—to object-oriented elements. To facilitate the presentation of this step, we will rely on a trivial banking loan process. This process, presented in Figure 3, has a single external task, called *approve transaction*. Despite its minimal size, it is able to illustrate the rules used by an OBPM to associate business process elements to object-oriented abstractions.

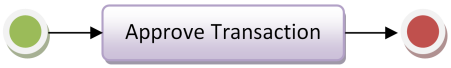

Fig. 3. Loan Process Definition

OBPMs provide three key abstractions for handling business processes:

- *Process Interfaces*, which provide means for information systems to delegate the business logic to business process engines. More specifically, process interfaces are used by information systems to trigger the execution of external tasks in the BPMS.
- *Data Classes*, which provide means for sharing data between information systems and business process engines.
- *Callback classes*, which provide means for business process engines to notify information systems about specific events or states. Callbacks are used mainly for adding external semantics to the execution of business processes (e.g., to validate credit cards in a process that depends on this information to proceed).

In the following subsections, these abstractions are presented in details, including the binding mechanisms provided by an OBPM to associate them to business process elements.

C. Process Interfaces

The most basic service that information systems require from a BPMS is the execution of external tasks. In a typical scenario, the end-user provides some information in a form, clicks submit, and generates an event that should be handled by the information system. The information system then delegates to the business process engine the execution of this external task, with the parameters provided by the user. Particularly, when using an OBPM, this delegation should rely on object-oriented abstractions denoting business process elements. Therefore, the first element that needs to be mapped is the process itself. When using an OBPM, a business process is represented by interfaces, called *process interfaces*, which define a contract between the information system and the BPMS.

Figure 4 illustrates the mapping of our running `LoanProcess` to a process interface. As showed in the figure, the methods in the process interfaces are associated to the process external tasks. Therefore, when the information system calls for example the `approveTransaction` method, the external task with the same name is executed by the BPMS. Moreover, the class that implements a process interface is provided at runtime by the OBPM, as discussed in Section IV. In this way, by using an OBMP, information systems become oblivious about the internal behavior of the BPMS.

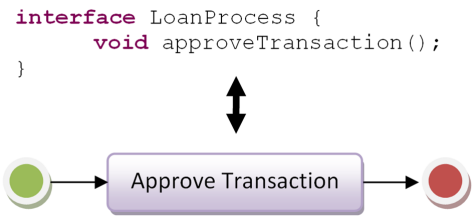

Fig. 4. Process interface example

D. Data Classes

Typically, a business process needs to manipulate global data [15, 16]. For example, in our running loan process, a possible data is the client identification. We propose that OBPMs should represent process data by key-value pairs. These pairs constitute the *process dataset*, and each entry in this dataset is called a *process attribute*. When a task is executed, it can access the process dataset to read or to write information. To represent the process dataset, a *data class* must be created, with fields denoting the key-value pairs in the dataset, as illustrated in Listing 3. In this example, the attribute `clientID` (line 2) is linked to the key-value pair in the business process whose key is `"clientID"`. An OBPM should guarantee that changes in the value of this attribute are reflected in the value maintained by the business process and vice-versa.

```
1  class LoanData {
2      String clientID;
3      String getClientID(){return clientID;}
4      void setClientID(String id){clientID = id;}
5  }
```

Listing 3. Data class example

To retrieve an object of a data class, an accessor method must be declared in the respective process interface. Listing 4 presents the `LoanProcess` interface with a method `getLoanData` (line 2) for accessing the process data class.

```
1  interface LoanProcess {
2      LoanData getLoanData();
3      void approveTransaction();
4  }
```

Listing 4. Process interface with a `getLoanData` accessor method

Besides abstracting the process dataset, an OBPM should provide means for representing the information handled by external tasks. More specifically, external tasks have two datasets: the *parameters* and the *results*. Parameters denote data from external entities that should be passed to the BPMS in order to execute particular tasks. Results are data produced by the execution of tasks, which must be returned to the task caller. To pass information to an external task, the method representing the task must declare the respective parameters. An example is presented in Listing 5, where the *approve transaction* task has now a parameter that represents the amount of money requested by the client.

```
1  interface LoanProcess {
2      LoanData getLoanData();
3      void approveTransaction(Number n);
4  }
```

Listing 5. Mapping task parameters to method parameters

External tasks can also return values to callers. Because methods in object-oriented languages usually cannot have multiple return values, a class should be created to represent the results. In our *approve transaction* task, a possible result is the *ID* of the transaction. The class representing the results—called `TransactionInfo`—and the updated `LoanProcess` interface are presented in Listing 6. When the `approveTransaction` method is executed, the result stored by the BPMS in the key named *transactionNumber* is automatically copied to the attribute `transactionNumber` of the `TransactionInfo` class (line 2).

```
1  class TransactionInfo {
2      Number transactionNumber;
3  }
4  interface LoanProcess {
5     LoanData getLoanData();
6     TransactionInfo approveTransaction(Number n);
7  }
```

Listing 6. Mapping the values returned by a task to the results of a method

E. Callback Classes

When tasks are processed by a 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, automated refactorings, etc. A preferred strategy is to allow the BPMS to callback services implemented by the information system. For this purpose, OBPMs provide support to *callback classes*, whose methods are automatically called when the tasks with the same name (not necessarily external tasks) are processed by the BPMS. Listing 7 illustrates the definition of a callback class for our loan process example.

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

Listing 7. Callback class example

It is important to highlight the differences between process interfaces and callback classes. Interfaces are used by client applications to execute business process external tasks. On the other hand, callbacks provide external semantics to any type of task. Someone may wonder why the client application does not call the callback methods directly, instead of using the process interface. Actually, the life cycle of a task may include many rules implemented by the business process engine. When a method from a process interface is called, the BPMS engine handles the request by executing internal services to accomplish the task. A possible internal service is to callback the application. As an example, an application might request the execution of a task that is not available. In this case, the BPMS will not advance the process or trigger the callback.

IV. Reference Architecture

In this section, we propose a reference architecture to guide the implementation of OBPM frameworks. As illustrated in Figure 5, this architecture has two main layers. The first layer, called *Workflow Connectivity* (WFC), provides an API to access the Abstract Business Process Model (ABPM), i.e., an API for handling process definitions, activities, etc. Moreover, this layer also provides means to connect an OBMP implementation to a concrete BPMS implementation. The second layer, called *Object-Workflow Mapping* (OWM), provides an API that represents, in terms of object-oriented abstractions, the elements of a business process. Therefore, information systems rely on abstractions provided by the OWM layer, which in turn access the services implemented by the WFC layer. Finally, the WFC communicates with a given BPMS implementation.

A. Workflow Connectivity Layer

The main objective of the Workflow Connectivity Layer is to shield OBPM users from manipulating low-level BPMS specific elements, therefore promoting independence of BPMS implementation. Listing 8 shows a code that relies on types exported by the WFC layer to perform the following actions:

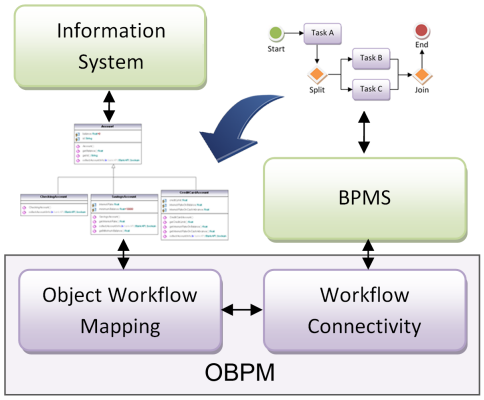

Fig. 5. OBPM reference architecture

to connect to a BPMS engine (lines 1-2), to create a process instance for the process named *myProcess* (lines 3-4), to start this process (line 5), to retrieve an activity instance named *myTask* (lines 6-7), and finally to execute this activity (line 8).

```
1 String url = "jwfc:bpmsX:res";
2 Session s = WorkflowManager.getSession(url);
3 ProcessInstance pi =
4         s.createProcessInstance("myProcess");
5 pi.start();
6 ActivityInstance ai =
7         pi.getActivityByName("myTask");
8 ai.complete();
```

Listing 8. WFC services to execute a process task

There are two types of relations with WFC interfaces. First, there are systems that call methods on these interfaces (as in Listing 8). Second, there are systems that implement the WFC interfaces, and act therefore as *driver implementers*. A driver should implement the abstract business process model defined in Section III. For example, consider an hypothetical BPMS X and its specific driver, called Driver X. In this case, the WFC layer exposes implementation independent interfaces, but relies on the driver classes to communicate with BPMS X.

Despite components to dispatch operations to a given BPMS, the WFC layer provides *application agents* that are used by the BPMS to call services from the information system. Such agents act as listeners and receive events from the BPMS. By intercepting these events, application agents can be used for example to implement functionality that must be executed when an activity is triggered (like callbacks).

B. Object-Workflow Mapping Layer

The Object-Workflow Mapping (OWM) layer enables the association of business processes to process interfaces, by implementing the mapping rules proposed in Section III-B. As illustrated in Figure 6, given a process interface `MyProcess`, the OWM layer is responsible for providing at runtime a class `MyProcessImpl` that implements this interface. This class relies on the WFC API, which in turn delegates the calls to a given BPMS implementation.

The OWM layer is also responsible for managing the application agents used to handle callbacks, as illustrated in Figure 7. First, the information system developer creates a callback class `MyProcessCallback` for a given process (step 1). The OWM is responsible for registering in the WFC layer an application agent to manage the callback methods in this class (step 2) and to install the corresponding hooks to execute this application agent (step 3). At runtime, the BPMS send events to the installed hooks (step 4), that in turn invoke the WFC layer (step 5). After that, the WFC calls the installed application agents (step 6). Finally, such agents invoke the corresponding method in the callback class (step 7).

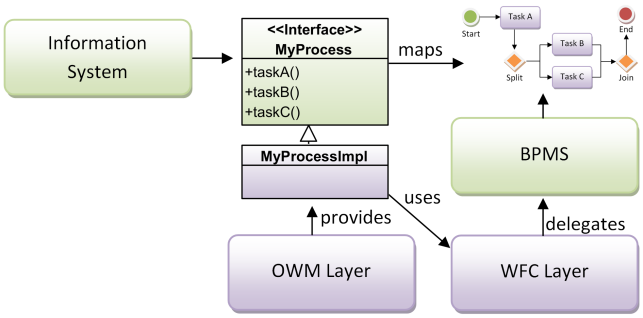

Fig. 6. OWM architecture

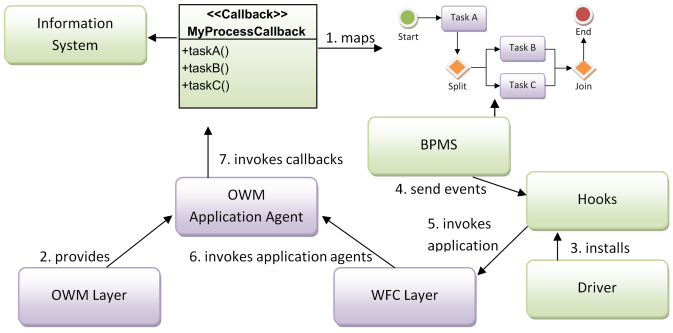

Fig. 7. Callback architecture

V. NextFlow: An Object-Business Process Mapping Framework

We implemented a real OBPM framework, called NextFlow, that follows the architecture described in Section IV. NextFlow was implemented in Java and makes use of many features of this language, like reflection and metadata annotations. Particularly, we used a convention based on names to map tasks and parameters to Java abstractions. For example, a task named *doIt* is mapped to the method `doIt` of the class that represents the process definition. Moreover, classes and interfaces are mapped to processes using annotations.

To implement process interfaces, NextFlow uses a Java API called Proxy[2]. Basically, this API allows the user to provide an interface and a listener object and it creates at runtime a proxy for this interface. The implementation of the proxy delegates its method calls to a listener provided by the API

[2]http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/Proxy.html

client. Therefore, the implementation of process interfaces are actually proxies that delegate their method calls to an specific listener provided by our implementation. This listener uses standard reflection features to read the metadata in the process interfaces and to execute the correct tasks in the BPMS.

Because the Proxy API only allows the creation of new classes based on interfaces, another mechanism was used to extend existing classes. When a class is referenced in a Java program, a component—called Class Loader [17]—is responsible for providing a reference to the class. A typical Class Loader searches the configured class path, and loads the bytecode stored in a class file into the virtual machine. It is possible to reference the Class Loader using objects of the type `ClassLoader`, which provides several methods for loading classes. One of them creates a new class from a parameter that is a byte array with bytecode instructions. It is possible to call this bytecode loading method explicitly, which is the strategy followed by our implementation. Our implementation generates the bytecodes representing the extended class and load them in the virtual machine using the `ClassLoader`. To generate bytecodes, we used a bytecode library called Cglib[3].

Finally, the proposed implementation depends on a driver to be fully operational. This driver is an implementation of the adapter design pattern to support the connection to different BPMSs. Currently, we implemented drivers for two BPMS, jBPM and Bonita.[4]

VI. Evaluation

To evaluate the OBPM framework concept proposed in this paper, we compared two implementations of the same system: (a) the first implementation relies on the native API provided by jBPM; (b) the second implementation relies on the abstractions proposed by NextFlow. In this comparison, we highlight the main difficulties faced by developers when using a native BPMS API and how NextFlow tackles such difficulties. The evaluation is divided in two parts. First, we present and compare the integration code required by both implementations. After that, we present a quantitative analysis that reveals how much effort can be saved by using an OBPM like NextFlow.

The evaluation relies on a system—named Charging System—that provides a mechanism for transferring money using cell phone messages. For example, consider two cell phone users, John and Mary. Suppose also that John needs some money from Mary. In this scenario, John can use his cell phone to send a credit transfer request message to Mary. If Mary authorizes the request, the charging system transfers the requested amount of money from Mary to John. Figure 8 shows the business process that describes the Charging System using jBPM process definition language. We will rely on this process to compare the implementations based on native jBPM access and using NextFlow. It is worth to mention that this process definition uses elements not available in the abstract business process model presented in Section III-A. For example, it relies on different types of splits. However, our OBPM framework proposal assumes an abstract model, and such specific elements are represented using the generic elements of this model.

A. OBPM Abstractions for Interfacing with the Charging System

This section presents the three abstractions proposed by our OBMP solution, which must be created for interfacing with the business process in Figure 8. First, when using an OBPM framework, a *process interface* must be created to represent the business process. Listing 9 shows the `ChargingProcess` interface that maps the charging business process and exposes its external tasks.

```
1  @Process("org.nextflow.example.payment")
2  interface ChargingProcess {
3    void requestPayment(String from, String msg);
4    void cancelProcess();
5    void sendAuthorizationResponse(String msg);
6    ChargingProcessData getData();
7  }
```

Listing 9. Process interface with external tasks

In this interface, there are three methods representing the external tasks `requestPayment`, `cancelProcess`, and `sendAuthorizationResponse` (lines 3–5). The parameters of the methods are declared as prescribed by the business rules. For example, to request a payment, it is necessary to inform who requested the payment and the message that contains the request (line 3). The last method returns an object that represents the data of the process (line 6). The `ChargingProcessData` *data class* is automatically recognized by NextFlow as the data structure of the process (as it is returned by a method with the prefix `get`). Listing 10 shows this class, which consists of a traditional POJO class, i.e., a class containing only attributes and their respective getters and setters.

```
1  class ChargingProcessData {
2    String from;
3    String to;
4    Integer value;
5    Boolean validRequest;
6    Boolean enoughCredit;
7    Boolean authorized;
8    //getters and setters
9  }
```

Listing 10. Data Class that represents the process data in NextFlow

The third component that must be created in the NextFlow implementation is the *callback class*, as presented in Listing 11. Basically, this class has a method for each task defined in the business process.

```
1  @Process("org.nextflow.example.payment")
2  class ChargingCallback {
3
4    ChargingProcessData data;
5
6    public void requestPayment(String from,
7                               String msg){
8      //provides the request payment behavior
9    }
10   public void checkCredit(){
11     //provides the check credit behavior
```

[3]http://cglib.sourceforge.net
[4]http://www.bonitasoft.com.

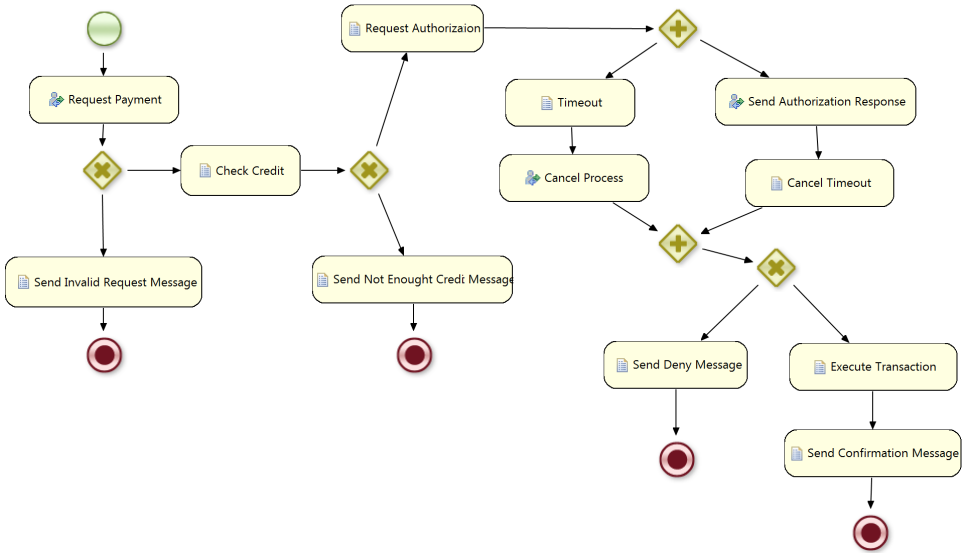

Fig. 8. Charging System

```
12     }
13     //other callback methods
14   }
```

Listing 11. Callback class for the Charging System using NextFlow

B. Comparison with jBPM Native API

In this section, we compare the implementations based on NextFlow and based on jBPM native API for supporting the following key actions when integrating our Charging System with a BPMS: (a) create a connection; (b) start a new process; (c) execute external tasks; (d) execute automatic tasks.

Creating a Connection: In order to send messages to a BPMS engine a connection must be created. In jBPM, this connection is represented by a `StatefulKnowledgeSession` interface, while in NextFlow it is represented by the `WorkflowObjectFactory` interface. Listing 12 shows the code that creates a jBPM `StatefulKnowledgeSession` and Listing 13 shows the corresponding code to create a NextFlow `WorkflowObjectFactory`.

```
1  StatefulKnowledgeSession kSession;
2  KnowledgeBuilder b = KnowledgeBuilderFactory
3                       .newKnowledgeBuilder();
4  Resource r = ResourceFactory
5               .newClassPathResource("cs.bpmn");
6  b.add(r, ResourceType.BPMN2);
7  KnowledgeBase kBase = b.newKnowledgeBase();
8  kSession = kBase.newStatefulKnowledgeSession();
```

Listing 12. Creating a session using the jBPM API

```
1  WorkflowObjectFactory factory;
2  String url = "jwfc:jbpm:cs.bpmn";
3  Configuration c = new Configuration(url);
4  c.addCallbackClass(ChargingCallback.class);
5  factory = c.createFactory();
```

Listing 13. Creating a session using NextFlow

A drawback of the jBPM implementation is the fact that it relies on a very specific BPMS API. In other words, there is not a standard interface to interact with the BPMS engine. Therefore, if someone needs to change the BPMS engine, the code in Listing 12 must be completely changed. On the other hand, this fact does not happen with NextFlow implementation. In this case, a change in the BPMS implies only in changing a URL (line 2, Listing 13).

Starting a New Process: The method that starts a new process is called `startNewChargingProcess`, presented in Listing 14 for the jBPM implementation. The created process is represented by a `ProcessInstance` object (lines 4–5). The `startProcess` method (line 5), besides the *process id*, receives a map of parameters, which are used by the BPMS to callback the information system.

```
1  ProcessInstance startNewChargingProcess() {
2    Map<String, Object> parameters = ...;
3    parameters.put("manager", ...);
4    ProcessInstance processInstance= ...;
5    return processInstance;
6  }
```

Listing 14. Starting a process using jBPM

To start a process in NextFlow, a `start` method is used, as showed in Listing 15. The object returned by this method provides access to business methods, as defined in the `ChargingProcess` interface. Therefore, it is easier to call tasks using this interface than using an specific BPMS API.

```
1  ChargingProcess startNewChargingProcess() {
2    return factory.start(ChargingProcess.class);
3  }
```

Listing 15. Starting a process using NextFlow

Executing External Tasks: Listing 16 shows the code that executes the *request payment* task. It checks whether the values

are correct (lines 2–4), creates the task parameters (lines 7–11), and then complete the work item (lines 12–13), which is an object that represents a task to be executed by jBPM.

```
1  void executeTask(String from, String msg,
2                           NodeInstance node){
3    Pattern p = Pattern.compile(...);
4    Matcher matcher = p.matcher(msg);
5    if(matcher.matches()){
6      String to = matcher.group(1);
7      String value = matcher.group(2);
8      Map<String, Object> res = new ...;
9      res.put("r_from", from);
10     res.put("r_to", to);
11     res.put("r_value", new Integer(value));
12     res.put("r_validRequest", true);
13     int wid = node.getWorkItemId();
14     kSession.getWorkItemManager()
15                 .completeWorkItem(wid, res);
16   } else {
17     Map<String, Object> res = new ...;
18     res.put("r_from", from);
19     res.put("r_validRequest", false);
20     int wid = node.getWorkItemId();
21     kSession.getWorkItemManager()
22                 .completeWorkItem(wid, res);
23   }
24 }
```

Listing 16. Executing an external task in jBPM

The equivalent code in the NextFlow-based implementation is presented in Listings 17 and 18. In Listing 17, the process interface is used to execute the task. Therefore, from the perspective of a developer that just wants to execute a task, this is the only code required.

```
1  void executeTask(String from, String msg,
2                   ChargingProcess p) {
3    p.requestPayment(from, msg);
4  }
```

Listing 17. Executing an external task in NextFlow

The actual behavior of the task is implemented in a method from a callback class, as presented in Listing 18. First, the code checks whether the message pattern is correct (lines 3–5); if it is correct, it sets the values in the corresponding process variables (lines 8–10); otherwise, it configures the process with an invalid request (line 12). Although in NextFlow the implementation is divided in two classes, it is actually more modular. Basically, there are two perspectives involved in this implementation: the perspective of a developer that wants some task to be executed (client) and the perspective of a developer that implements the task behavior (provider). The first developer does not need to know how the task is implemented, which is a requirement that is not fulfilled by jBPM implementation. In fact, it is possible to partition the jBPM code just like in NextFlow, but it would add extra complexity to the design.

```
1  void requestPayment(String from, String msg){
2    data.setFrom(from);
3    Pattern pattern = Pattern.compile(...);
4    Matcher matcher = pattern.matcher(msg);
5    if(matcher.matches()){
6      String to = matcher.group(1);
7      String value = matcher.group(2);
8      data.setTo(to);
9      data.setValue(new Integer(value));
10     data.setValidRequest(true);
11   } else {
12     data.setValidRequest(false);
13   }
14 }
```

Listing 18. Callback associated to an external task in NextFlow

Another feature of the NextFlow implementation is the support for static type checking. For example, instead of writing `results.put("r_to", to)`, NextFlow supports a code like `data.setTo(to)`.

Executing Automatic Tasks: The *check credit* task is an automatic task, i.e., it is executed by the business process engine. Therefore, there is no need to call this task explicitly. Listing 19 shows the code of the callback that handles this task in the jBPM implementation. Listing 20 shows the code for the NextFlow implementation. In this implementation, the `checkCredit` method is implemented in a callback class.

```
1  void executeScriptTask(NodeInstance node) {
2    WorkflowProcessInstance p =
3                        node.getProcessInstance();
4    Integer val = (Integer)p.getVariable("value");
5    String to   = (String) p.getVariable("to");
6    Integer credit =
7                  chargingManager.getCreditFor(to);
8    boolean enoughCredit = credit >= val;
9    p.setVariable("enoughCredit", enoughCredit);
10 }
```

Listing 19. Callback associated to an automatic task in jBPM

```
1  void checkCredit(){
2    Integer credit = chargingManager
3                        .getCreditFor(data.getTo());
4    data.setEnoughCredit(
5                         credit >= data.getValue());
6  }
```

Listing 20. Callback associated to an automatic task in NextFlow

C. Quantitative Analysis

Table II summarizes the effort required to implement the Charging System, in terms of lines of code (LOC), number of characters, number of classes, and number of import statements, regarding only API packages. The presented values were collected using a small Java application we implemented. We can observe that NextFlow implementation required 30% less lines of code and 50% less characters. Finally, jBPM implementation requires more classes (17 classes, against 11 classes when using NextFlow). This fact happened because jBPM demands the implementation of auxiliary classes to support most of the integration actions.

TABLE II
SIZE OF BOTH IMPLEMENTATIONS

Metrics	jBPM	NextFlow	Delta
LOC	475	330	-30%
Char Count (Kb)	25	12	-50%
Classes	17	11	-35%
API Imports	63	6	-90%

This reduction in size and complexity is particularly important in the implementation scenarios where the use of OBPMs are recommended. Basically, OBPMs are useful when the BPMS is one of the components of an enterprise software architecture and therefore code is required to support its integration with the remaining components in this architecture.

D. Threats to Validity

In this section, NextFlow was evaluated by means of a comparison with an implementation based on direct BPMS access. The comparison with just one BPMS potentially limits the validity of our findings, as other BPMSs may have different features and characteristics. Nevertheless, even if we regard NextFlow as a solution specific to jBPM, our work at least shows that it is possible to map business process elements to high-level object-oriented abstractions, which is the main contribution of this paper. Moreover, the problems appointed in the jBPM implementation are also present in other BPMSs. For example, in YAWL process variables are declared in a similar way as in jBPM [18]. Another threat to validity is that both the Charging System and the NextFlow framework were implemented by us. Therefore, this fact might have favored our evaluation, because one can argue that a business process suitable for NextFlow was used. In our defense, we argue that all process elements provided by jBPM are present in the Charging Process.

VII. Conclusion

In this paper, we presented an approach to integrate business process management systems (BPMS) and enterprise software architectures, by means of what we called an Object–Business Process Mapping (OBPM) framework. We claimed that OBPMs can contribute to a broader adoption of BPMS, because they do not represent a disruptive technology regarding current enterprise software architectures, frameworks, and libraries. We also proposed a reference architecture for implementing OBPM frameworks and a concrete implementation of this architecture, called NextFlow. We compared two implementations of an information system, using NextFlow and a native BPMS API. We showed that the OBPM-based implementation is structured around high-level object-oriented abstractions, including process interfaces, data classes, and callback classes.

Currently, our prototype implementation is available only for Java-based information systems. Moreover, we only provide driver support to two BPMS (jBPM and Bonita). In the near future, we plan to support new languages and BPMSs. We also plan to work on a qualitative assessment with real developers and on a model-driven extension of our current solution. NextFlow is publicly available for download at: `http://nextflow.org`.

Acknowledgments

This research is supported by grants from FAPEMIG and CNPq.

References

[1] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley, 2003.

[2] R. Terra and M. T. Valente, "A dependency constraint language to manage object-oriented software architectures," *Software: Practice and Experience*, vol. 32, no. 12, pp. 1073–1094, 2009.

[3] W. Aalst, "The application of Petri nets to workflow management," *Journal of Circuits, Systems, and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[4] P. Lawrence, Ed., *Workflow handbook*. John Wiley & Sons, 1997.

[5] M. Dumas, M. Rosa, J. Mendling, and H. Reijers, *Fundamentals of Business Process Management*. Springer, 2012.

[6] P. Muth, J. Weibenfels, M. Gillmann, and G. Weikum, "Integrating light-weight workflow management systems within existing business environments," in *15th International Conference on Data Engineering (ICDE)*, 1999, pp. 286–293.

[7] F. P. Brooks, "No silver bullet – essence and accidents of software engineering," *IEEE Computer*, vol. 20, no. 4, pp. 10–19, 1987.

[8] M. Havey, *Essential business process modeling*. O'Reilly, 2005.

[9] OMG, "BPMN - Business Process Model and Notation Version 2.0," 2011.

[10] D. Manolescu, "Micro-workflow: A workflow architecture supporting compositional object-oriented software development," Ph.D. dissertation, Univ. of Illinois, 2001.

[11] WfMC, "Workflow Management Application Programming Interface (Interface 2 & 3) Specification," 1999.

[12] L. Borgonon, M. Barcelona, J.Garcia-Garcia, M. Alba, and M. Escalona, "Software process modeling languages: A systematic literature review," *Information and Software Technology*, vol. 56, no. 2, pp. 103 – 116, 2014.

[13] W. Aalst, "Three good reasons for using a petri-net-based workflow management system," in *Information and Process Integration in Enterprises*, 1996, pp. 179–201.

[14] E. Borger, "Approaches to modeling business processes: a critical analysis of BPMN, workflow patterns and YAWL," *Software and Systems Modeling*, pp. 1–14, 2011.

[15] W. Aalst and K. Lassen, *Translating workflow nets to BPEL*. Research School for Operations Management and Logistics, 2005.

[16] P. Reimann, H. Schwarz, and B. Mitschang, "Design, implementation, and evaluation of a tight integration of database and workflow engines," *Journal of Information and Data Management*, vol. 2, no. 3, pp. 353–368, 2011.

[17] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*, 4th ed. Addison-Wesley, 2005.

[18] W. Aalst, L. Aldred, M. Dumas, and A. Hofstede, "Design and implementation of the YAWL system," in *16th Conference on Advanced Information Systems Engineering (CAiSE)*, vol. 3084, 2004, pp. 142–159.