Towards Process-Oriented Modelling and Creation of Multi-Agent Systems

Tobias Küster, Axel Heßler, and Sahin Albayrak

DAI-Labor, Technische Universität Berlin, Germany
`tobias.kuester@dai-labor.de`

Abstract. Different ways of integrating business processes and agents have been proposed, but using restricted process models or targeting only single agents, none of them is truly convincing. Nevertheless, business processes have many notions in common with agents and would be well suited for modelling complex multi-agent systems. In this paper, we combine concepts of two existing approaches to a mapping from business process diagrams to readily executable agent components. The results are well-structured and extensible, and at the same time account for nearly the entire expressiveness of the process modelling notation.

Keywords: Technological, Methodological.

1 Introduction

In recent times, different approaches for modelling agents and multi-agent systems using business process diagrams and related notations have been introduced (e.g., [6], [10], [18]). However, none of these approaches is really compelling. Often, very simple workflow models are used, or if a more expressive process modelling notation is chosen, then only a limited subset of the language is covered. Furthermore, usually only single agents are targeted, while interactions between agents – which could very well be modelled using many process notations – are not regarded.

This is unfortunate, since process diagrams share many concepts and abstractions with multi-agent systems – in particular sophisticated notations such as the Business Process Model and Notation (BPMN) [22]. Those notations can be used for modelling the intertwined workflows of different participants in a process, as well as their interactions and communication, or their reactions to external events. The focus lies much more on *what* has to be done and less on *how* it is implemented. Thus, despite the shortcomings of existing approaches, BPMN and related notations appear to be very well suited for modelling agents and particularly multi-agent systems.

In this paper we take a look at some of the existing approaches – particularly the WADE extension to the JADE agent framework [10], and a mapping from BPMN to the agent-oriented scripting language JADL [18] – and combine the strong sides of both into a new approach. The result is a mapping from BPMN diagrams to behaviour components for the JIAC multi-agent framework [19].

In this way, the core components of the agents can easily be modelled with and generated from BPMN process diagrams. Thus, we are helping to close the gap between design and implementation of multi-agent systems [8]. The resulting Java classes are similarly structured and as extensible as those of WADE, but they exhibit the expressiveness of BPMN, including communication between agents and event-handling, both as part of the workflow and for triggering the process.

The remainder of this paper is structured as follows: First, we discuss some related work, most notably the WADE framework and the mapping from BPMN to JADL, with their benefits and shortcomings. Then, in Section 3, we take a closer look at BPMN and the JIAC framework, and how they fit together, Thereafter, we describe how BPMN processes can be mapped to semantically equivalent JIAC Agent Beans (Section 4), and how the transformation was implemented (Section 5). In Section 6, the mapping is illustrated using an example, before we finally wrap up and discuss our results.

2 Related Work

Different approaches for combining process modelling and agent-oriented software development have been devised. Some using BPMN, others using simpler notations; some using code generations, others employing interpreting approaches. Each of those have their strengths and weaknesses.

In the following we discuss several works that are highly relevant to the approach described in this paper: The original mapping from BPMN to BPEL, a mapping from BPMN to JIAC's scripting language JADL, the WADE framework, mapping workflows to JADE behaviours, and GO-BPMN, a combination of BPMN and goal hierarchies.

2.1 Transformation from BPMN to BPEL

One of the motivations for developing BPMN was to provide a standardised graphical notation for *BPEL*, the Business Process Executable Language. Consequently, a mapping from BPMN to BPEL is part of the BPMN specification [22, Chapter 14], and a number of alternative or extended mappings have been proposed by various other authors (see for example [20], [23]).

In many aspects, the mapping is very straightforward: Each pool is mapped to a BPEL process (which can be deployed as a Web service), and the several events and activities within are mapped to the workflow of the process. The process is made up mostly of Web service calls, assignments and flow control, but can also contain, e.g., event handling based on timing and incoming messages. Given a sufficiently detailed BPMN diagram, the resulting BPEL process can be readily executable.

Still, there are enough elements in BPMN for which no mapping to BPEL is given. Thus, while BPMN was created with the mapping to BPEL in mind, it is not just a visualisation for BPEL but a distinct, self-contained language – and in fact more expressive than BPEL itself. Among the elements that are not mapped to BPEL are somewhat obscure elements such as the *ad-hoc* subprocess, or the complex gateway, but also many types of events and tasks.

2.2 Transformation from BPMN to JADL

In prior work of mapping BPMN to agents [14], JIAC's service-oriented scripting language *JADL* [15] was used as the target of the transformation.

Being conceptually close to BPEL, the mapping is similar, and the process can be mapped very directly to different language elements of JADL. For instance, like BPEL, JADL has dedicated language elements for complex actions such as invoking other services, or for sending and receiving messages, making the generated code compact and easy to comprehend.

Each pool in the BPMN process is mapped to a JADL service, and the service's input parameters and result types are derived from the pool's start- and end events [18]. Further, for each start event, a Drools rule is created, starting the respective JADL service on the occurrence of the given event (e.g., an incoming message, or a given time). Also, for each participant in the BPMN process, an agent configuration file is created, setting up the individual agents, each equipped with an Interpreter Bean and Rule Engine Bean, together with the generated JADL services and Drools rules.

Alternatively, the JADL services and rules created from the BPMN processes can be added to a running JIAC agent, thus dynamically changing its behaviour.

2.3 WADE: Workflows for JADE

A different approach, from which some of the concepts in this work have been drawn, is *WADE (Workflows and Agents Development Environment)*, which is an extension to the JADE multi-agent framework [3]. Using WADE, certain aspects of the behaviour of a JADE agent can be modelled using a simple workflow notation [10,9]. The workflows basically consist of only two elements: Activities and Transitions.

Using the *Wolf* tool [11], JADE behaviour classes can be generated from those workflow models. The generated Java classes show a clear distinction between the workflow (the order of the activities, together with conditions and guards) and the several activities. Each of them is mapped to an individual Java method that can either refer to existing functionalities or be implemented by the developer. Using this separation, generated workflows can safely be altered or extended.

However, the expressiveness of WADE is restricted by the simplistic workflow notation, which allows only the most basic workflows to be modelled. While the transitions can be annotated with guards (conditions), it seems impossible to model parallel execution and synchronisation, let alone more advanced concepts such as event handling or messaging. In fact, each workflow diagram covers only the behaviour of an isolated agent; to our knowledge, interactions between agents can not be modelled.

Later, WADE has been extended to provide better support for long-running business processes, event handling, user-interaction and Web-service integration [5,4] and as of today appears to be a very mature product used in many projects.

2.4 GO-BPMN and Go4Flex

In *GO-BPMN* (Goal-oriented BPMN), process models are combined with a goal-hierarchy and executed by agents [12]. The authors highlight the high flexibility of the system, and the prospects of parallelisation, but they also write that testing the system is difficult due to possible side-effects of the processes regarding other goals [7].

The individual processes (the “leafs” in the goal hierarchy) are described as BPMN processes; however, only a subset of BPMN is used. Particularly, each diagram shows only a single pool, and thus, as in the case of WADE, no communication and interaction can be modelled, but just the behaviour of a single agent. While using goals for connecting the individual processes is quite promising, in our opinion process diagrams can more efficiently be used at a higher level of abstraction, e.g., for providing an overview of the system as a whole, instead of for isolated behaviours of individual agents.

A similar approach is *Go4Flex*, or *GPMN* [6]. Like GO-BPMN, Go4Flex uses goal hierarchies with BPMN processes being the leafs. Both the goals and the processes are interpreted by Jadex agents [25]. The authors also present a mapping from FIPA/AUML interaction diagrams [2] to BPMN processes [24].

2.5 Other Approaches

While those are the works most similar to our own, there are of course other, slightly different approaches, that shall not go unmentioned.

Agent UML, or AUML as already mentioned above, extends the UML with several agent-specific diagram types, most prominently interaction diagrams [2]. However, while serving very well for describing the interactions among agents, interaction diagrams – following the principles of UML – show just this single aspect of multi-agent systems. BPMN diagrams, on the other hand, can be seen as a combination of AUML interaction and activity diagrams and thus seem to be better suited for conveying the whole picture of the behaviours and interactions.

In another approach, multi-agent systems are modelled as ‘electronic institutions’ [27], describing their common ontologies, roles, interactions and norms. Those norms are monitored and enforced by the agent runtime, facilitating the operation of open systems, where agents might try to break those rules. Similar to this, in 2COMM, interaction protocols are represented as artefacts, not only encapsulating the different roles and commitments involved in the interaction, but also providing for functionalities such as logging, auditing, etc. [1].

Finally, there are numerous agent development methodologies, many of which also make use of sophisticated graphical notations for one end or another. One of those is i^*, which is used in the TROPOS Methodology, among others [30].

The focus here lies particularly on modelling the *social* relationships of the several actors involved in the systems: Their goals, intentions, and mutual 'strategic dependencies'. While *i** itself is not used for modelling processes, it could well be used complementary to, e.g., BPMN to model the rationale behind the agents' behaviours and interactions.

3 A Closer Look at BPMN and JIAC

As we have seen, there are numerous approaches for combining process modelling and multi-agent system engineering, but to the best of our knowledge none of them makes full use of the expressiveness of BPMN or a similarly powerful process notation. This is unfortunate, since BPMN provides many notions that could very well be used for modelling high-level multi-agent behaviour.

In the following, we will take a closer look at the BPMN language and the JIAC agent framework, being the domain and co-domain of the mapping discussed in the next section of this paper.

3.1 BPMN

The *Business Process Model and Notation* [22] is a workflow representation that can be used both as a description language for real-world processes, and as a high-level modelling language for computer programs. It can be seen as a combination of UML's Activity Diagrams and Sequence Diagrams, depicting both the actors' internal processes and their interactions. An example diagram is shown in Figure 1.

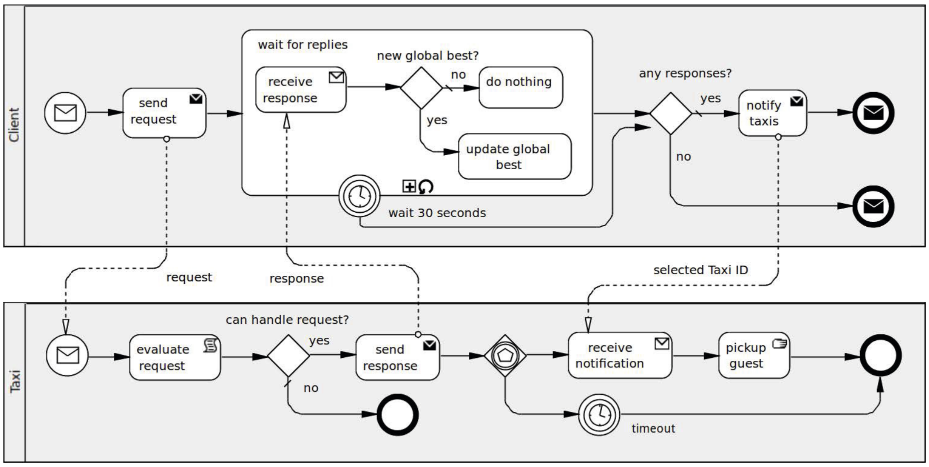


Fig. 1. Example BPMN Diagram: Taxi Request Service

BPMN diagrams can be understood at three levels of abstraction:

1. The diagrams are made up of a few easily recognisable elements, i.e. *events* (circles), *activities* (boxes) and *gateways* (diamonds), connected by *sequence-* and *message flows* and situated in one or more *pools*.
2. These basic elements are further distinguished using sets of marker icons, e.g., *message*, *timer*, and *error* events, or *parallel* and *exclusive* gateways.
3. Each element features a number of additional attributes that are hidden from the diagram and contain most of the information that is necessary for automated code generation, e.g., properties and assignments.

Consequently, the essence of a BPMN diagram is easily understood by all business partners, including those who have great knowledge in their domain but little understanding of programming and multi-agent systems. At the same time, BPMN diagrams provide enough information for the generation of executable programs.

A variety of notational elements make BPMN diagrams well suited for the design of distributed systems in general and multi-agent systems in particular. The process diagrams are subdivided into pools, each representing one participant in the process. Using message flows for communication between pools, even complex interaction protocols can be modelled clearly. Further, the notation supports features such as event- and error handling, compensation, transactions and *ad-hoc* behaviour.

In fact, one could argue that BPMN is *too* expressive, featuring many elements that are rarely used in practice [21] as well as redundancies w.r.t. how certain concepts can be modelled. Also, the semantics of some elements of BPMN – particularly those not covered in the official mapping from BPMN to BPEL [22, Chapter 14] – are not very clearly defined; however, there is an increasing number of approaches describing the semantics of BPMN using, e.g., Petri nets [13], and version 2.0 of the specification made things clearer, too.

The reason why Petri nets are not used in the first place is: While Petri nets have very clear semantics, and basically everything can be expressed as a Petri net, some high-level constructs that are directly supported by BPMN (e.g., event handling and cancellation) would require huge, incomprehensible Petri nets. Thus, while Petri nets are well suited for the formal specification of a workflow, they are not the best choice for modelling.

BPMN is neither the first process modelling notation, nor will it be the last. However, given its high level of adoption in practical process modelling [26], it has proven to be a good choice for modelling distributed computing systems, combining a high-level overview of the system with all the necessary details about its implementation and execution.

3.2 JIAC

JIAC V (Java-based Intelligent Agent Componentware, version 5) is a multi-agent development framework and runtime environment [19]. Among others, JIAC features message-based inter-agent communication, tuple-space based agent memory, transparent distribution of agents and services, and provides support for dynamic reconfiguration in distributed environments, such as component exchange at runtime. Individual JIAC agents are situated within Agent Nodes, i.e. runtime containers, which also provide support for migration. The agents' behaviours and capabilities are defined in a number of so-called *Agent Beans* that are controlled by the agent's life cycle. The different structures and elements of a JIAC multi-agent system are shown in Figure 2.

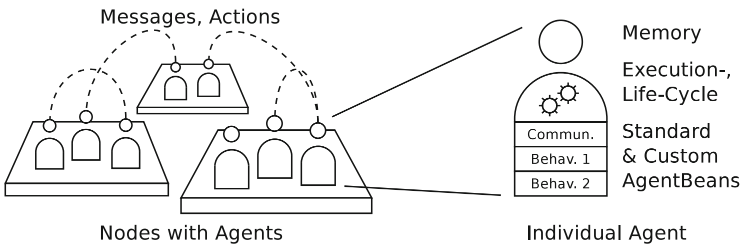


Fig. 2. Components of a JIAC multi-agent system and individual agents

Each JIAC agent is equipped with a *Communication Bean*, allowing agents to send and receive messages to and from other agents or groups of agents (multi-casting to message channels). The messages are not restricted to FIPA[1] messages and can have any serialisable data as payload. Other commonly used Agent Beans are the *Rule Engine Bean*, integrating a Drools[2] rule engine into the agent's memory for reactive behaviour, and the *Interpreter Bean*, providing an interpreter for the service-oriented scripting language JADL [15].

Besides these and other predefined Agent Beans, the developer is free to add application-specific Beans to the agent. Each such Agent Bean can:

- implement a number of *life-cycle* methods, which are executed when the agent changes its life-cycle state, such as initialized, or started,
- implement an *execute*-method, which is called automatically at regular intervals once the agent is running (i.e. cyclic behaviour),
- attach *observers* to the agent's memory, being called, e.g., each time the agent receives a message or its world model is updated, and
- contribute *action*-methods, or services, which are exposed to the directory and can be invoked by other agents or other Beans of the same agent.

Using these four mechanisms, it is possible to define all of the agents' capabilities and behaviours. For details on programming JIAC Agent Beans, we refer readers to the JIAC Programmers' Manual [16].

[1] Foundation for Intelligent Physical Agents: `http://www.fipa.org/`
[2] JBoss Drools: `http://www.jboss.org/drools/`

4 A Mapping from BPMN to JIAC Agent Beans

While the mapping from BPMN to JADL is well suited for modelling high-level behaviour or services, traditional JIAC Agent Beans were still advantageous – and often necessary – for defining the better part of the agent's behaviour, for instance when it comes to the integration with user interfaces or external libraries. Consequently, complementary to the mapping to JADL, a mapping to JIAC Agent Beans was developed [28].

The mapping is conceptually close to WADE: Each Pool in the BPMN diagram is mapped to one Agent Bean, i.e. a Java class, with one method for the workflow, and one method for each individual activity of the process.[3] The *workflow method* acts as an entry point to executing the process, while the several *activity methods* are invoked by the workflow method in accordance with the ordering of the activities in the process. The different workflow agent beans created in this way for the several pools representing one participant then make up the behaviour of the respective agent role.

Table 1 shows a high-level overview of the mapping. In the following, we will describe the several aspects of the mapping in detail. Finally, we will briefly illustrate how process modelling can be integrated into the overall development method.

Table 1. Overview of Mapping from BPMN to Agent Beans

BPMN Element	Agent Concept
participant	agent role (implicit, not created)
pool	workflow agent bean, holding all of the below
workflow	structured workflow method
start events	mechanisms to trigger workflow method
tasks	activity methods, doing the actual work
subprocess	nested class, same structure as workflow bean
boundary events	event handler threads, interrupting the activity
properties	variables, in appropriate scope

4.1 Workflow Method

The workflow method is made up of calls to several activity methods, being arranged into sequences, if-else statements and loops. While this requires the process to be structured properly (see Section 5), the result is structured and understandable, resembling manually written code, i.e. using conditions and loops instead of `goto`-like successor-relations. Thus, if necessary, the generated code can still be easily extended or altered by hand.

[3] In the following, we will use the term “workflow” for the order the individual activities are executed in the process, and the term “process” for the whole ensemble of activities and their ordering, events, variables, etc.

At the same time, BPMN allows for much more expressive workflows to be modelled, compared to the rather minimalistic workflow notation used in WADE. In particular, the following concepts of BPMN are covered by the mapping:

– Parallel execution (BPMN's AND-Gateway) is mapped to multiple threads being started and joined.
– Subprocesses (composite activities) are mapped to internal classes following the same schema as the main class, with workflow- and activity methods for the activities embedded into the subprocess.
– Event handler (intermediate events attached to an activity) are also mapped to threads, running concurrently to the thread executing the activity itself, and interrupting this thread in case the respective event occurs.
– The same pattern is applied to event-based XOR-gateways; in this case the main thread will wait until one of the events has been triggered.

4.2 Properties and Assignments

BPMN specifies a number of non-visual attributes, such as properties (i.e. variables) and assignments. Properties can be declared in the scope of whole processes or individual activities (both atomic tasks and composite subprocesses). When declared in the scope of a process or subprocess, the property is visible to all elements (transitively) contained therein.

Accordingly, properties are mapped to Java variables in different scopes in the Agent Bean, reflecting their visibility in the BPMN diagram. Properties of the process are mapped to variables in the scope of the Agent Bean class, properties of a subprocess to variables in the scope of the embedded subprocess class, and properties of an activity to local variables in the scope of the activity method.

Assignments are always bound to an activity or event, and are included in the respective activity method. In BPMN, assignments can have an *assign-time* of either 'before' or 'after', determining whether the assignment has to be applied before or after the actual activity is executed (see below).

4.3 Activity Methods

The several activity methods have neither parameters nor a return value and always follow the same schema:

1. *Properties*: First, for each property in the scope of the activity one Java variable is declared, using the respective data type.
2. *Start Assignments*: Then, assignments of the activity with assign-time 'before' are applied, e.g., for setting the input parameters of a service call.
3. *Activity Body*: Now, the code corresponding to the actual activity is carried out, e.g., invoking a service, sending a message, or executing a user-defined code-snippet.
4. *End Assignments*: Finally, assignments with assign-time 'after' are applied, e.g., for binding the return value of a service call to a local variable.

5. *Loop*: If the activity's *loop* attribute is set, the content of the activity method is repeated in a loop as long as a given condition is satisfied.

Similar to the mapping to JADL, we can make use of JIAC's communication infrastructure, by mapping *message* events and *send* and *receive* tasks to sending and receiving JIAC messages, while *service* tasks are mapped to the invocation of a JIAC action (i.e. a service). *Script* tasks allow the developer to attach a custom snippet of Java code to the task. Further, *timer* events are mapped to a temporary suspension of the execution.

There are more types of tasks and events in BPMN, for which no mapping has been devised yet, but these are the most common and important ones. Elements that will be covered in the near future include the *rule* event, evaluating a given Java condition, as well as the *user* task, presenting a generic input dialogue to the user.

4.4 Event Handler

As mentioned above, event handlers (i.e. intermediate events attached to an activity's boundary) are mapped to threads running in parallel to the actual activity, interrupting it in case the event has been triggered. To realise this behaviour, the activity itself is wrapped in another thread, and a reference is passed to the event handler thread, running in a loop and periodically checking whether the respective event has occurred (e.g., whether a message has arrived, or whether a given time has passed). If so, a marker flag is set and the activity thread is interrupted.

In the workflow method, both threads are started, and the activity thread is joined. Finally, when the activity has been completed or aborted, the event handler thread is stopped and the workflow is routed accordingly to whether the event handler has been triggered or not.

4.5 Start Events and Starter Rules

Finally, the processes' start events have to be mapped to mechanisms for starting the process on the occurrence of the respective events. In the mapping to JADL, a number of Drools rules are created for this purpose. Using Agent Beans, these 'starter rules' can be integrated directly into the code, making use of the mechanisms introduced in Section 3.2.

- If the process has a start event with unspecified type, or *none* type, then the workflow method is invoked in the Agent Bean's `doStart()` method (one of the *life-cycle* methods), being called when the agent is started.
- For a *timer* start event, the Agent Bean is given an `execute()` method, regularly checking the current time against the time the process was last started, invoking the workflow method at a given time or interval.
- A *message* start event results in a message observer being attached to the agent's memory when the Agent Bean is started, which will then invoke the workflow method every time a matching JIAC message is received.

– Finally, in case of a *service* start event, the workflow method is marked with the annotation `@Expose`, exposing the workflow method as a JIAC action to be discovered and invoked by other agents.[4]

Besides creating these mechanisms, a *service* start event also results in the workflow method's input parameters being updated to correspond to the specified service parameters. Analogously, a *service* end event results in the workflow method's return value being set accordingly.

4.6 Development Method

In previous work, we presented a method for integrating process modelling into the overall multi-agent system development cycle [18], as shown in Figure 3. While this was aimed at the mapping from BPMN to JADL, most of the ideas and concepts can be carried over to the mapping to JIAC Agent Beans as well.

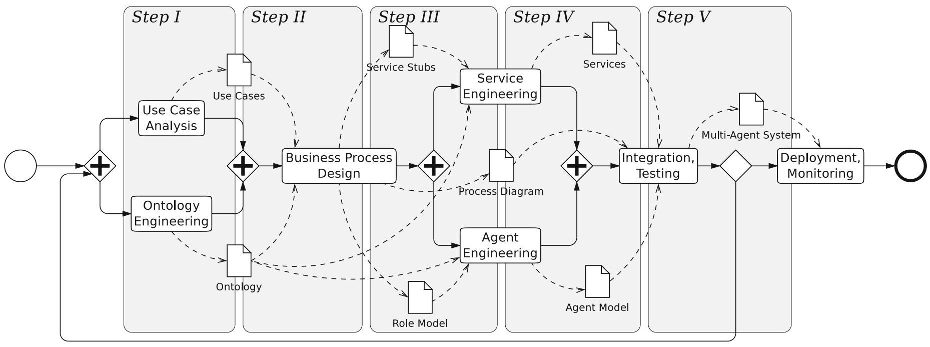


Fig. 3. Integration of process modelling into development method [18]

In a nutshell, we see process modelling as the next step after use case analysis. For each of the previously identified use case diagrams, one BPMN process diagram is created, holding one pool for each of the actors involved in the respective use case. Those diagrams should describe the behaviour and particularly the interaction of the several roles at a relatively high level of abstraction, illustrating the system behaviour without cluttering the diagrams with algorithmic details. The mapping then translates the pools to behaviours, encapsulated into Agent Beans, while each of the actors corresponds to a different agent role exhibiting those behaviours. Next, the generated JIAC Agent Beans can be extended with additional code not suited for inclusion in the process diagrams, and the agent roles are aggregated to concrete agents and the multi-agent system is set up.

[4] There is, as such, no *service* start event in BPMN. We use this term to distinguish *message* start events, where the message is in fact a service request.

5 Implementation

The first version of the mapping was implemented in the course of a diploma thesis [28] as an extension to the BPMN editor *VSDT (Visual Service Design Tool)*. The VSDT was developed with the goal in mind, to provide transformations from BPMN to diverse executable languages [17]. It also allows for the import of existing services, simulation/interpretation of process diagrams, and the generation of descriptive texts in written English from the process. Besides being a BPMN editor, it can also be used for creating the use case diagrams for connecting the different process diagrams that make up the entire system.

For exporting BPMN diagrams into different target languages, the VSDT uses a generic transformation framework [17]. The process can be subdivided into several stages, being executed one after the other:

1. Validation and Normalization: Check validity of BPMN diagrams and bring diagram into 'normalized' form to facilitate later stages.
2. Structure Mapping: Use pattern-matching to identify different structures, such as blocks and loops, and bring the diagram into a tree-like form.
3. Element Mapping: Tree-traversal of the structured process, performing the actual mapping to the target language (JIAC, JADL, BPEL, etc.)
4. Clean Up and Storage: Clean up generated code, merge with existing files, if any, write to output directory.

The first steps in mapping BPMN to Agent Beans – or any structured programming language – is to structure the process graph to a tree of sequences, decision blocks, loops, etc. [20]. To this end, a number of pattern matching rules are used, identifying different structures in the workflow and substituting them with dedicated structural elements. This functionality is provided by the VSDT's transformation framework and can be reused for the different target languages [17]. Thus, only the actual mapping of individual process elements to fragments of Java code, as specified in the previous section, had to be implemented.

This element mapping has been separated into two stages. First, the structured process model is translated to an intermediate model, being a high-level representation of the structure of a JIAC Agent Bean. This is done by traversing the process model, which now has a tree-structure, and thereby creating and assembling the respective elements of the Agent Bean model. Then, this model can be translated straightforwardly to executable Java code using a number of templates for the JET framework.[5] Using JET and JMerge, parts of the generated Agent Bean code can safely be modified and merged in case the process model changes and has to be re-generated.

[5] JET (Java Emitter Templates) is part of the Eclipse Model To Text (M2T) project: `http://www.eclipse.org/modeling/m2t/`

6 Example

In this section we will illustrate several aspects of the mapping by means of the simple example diagram from Section 3, shown in Figure 1.

The BPMN diagram consists of two pools, each representing an agent role: *Client*, and *Taxi*. The client's process is exposed and started as a service, expecting a customer ID, current location, desired destination and time of arrival, and returning the ID of the taxi selected for the tour, if any.

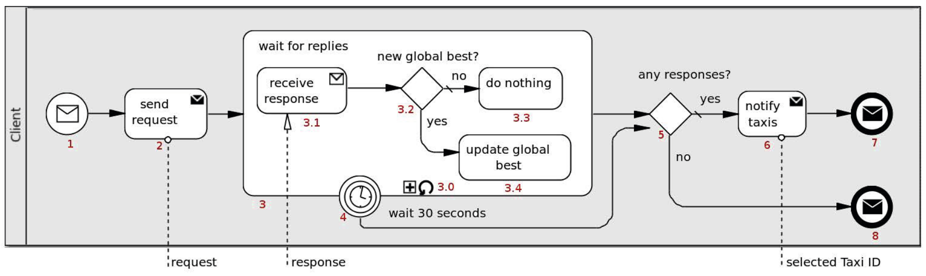
The interaction between the two starts by the client sending a request (customer ID, location, destination, desired time of arrival) to all available taxis, which evaluate the request and decide whether to accept it. If so, they send a response (taxi ID, estimated time of arrival, price) back to the client. Meanwhile, the client enters a looping subprocess, listening to responses and memorising the best response, until after 30 seconds the subprocess is interrupted by the attached timer event. The client then sends a notification to the selected taxi. The taxis listen to incoming message, either preparing to pick up the guest if the notification is received, or ending the process after waiting for a few more seconds. Note that the several properties (variables) and assignments are not visible in the diagram.

The resulting Agent Bean for the *Client* role is shown in Figure 4, along with the client's part of the process diagram for reference. The entire code was automatically generated and only slightly shortened to improve readability and to better fit into the figure. The full code also contains JavaDoc comments (not shown here) with descriptions of the bean class and each of the activity methods, taken from the *description* attribute of the respective BPMN elements.

As can be seen, the control-flow of the process is reflected in the `workflow()` method, which is also exposed as a JIAC action, or service. The workflow method is dominated by the threads for running the subprocess and the attached event handler, but also contains an if-else-statement for the gateway at the end of the process. The activities *send request* and *notify taxis* are mapped to two similar methods for sending JIAC messages to the specified message groups.

The code for, e.g., sending and receiving messages is quite extensive, and there are several components, such as the event handler classes, that are needed again and again for different workflows. Consequently, these parts are provided by the superclass `AbstractWorkflowBean`, allowing the generated code to be much more compact and readable.

The subprocess is mapped to the inner class `WaitForReplies_Sub`, also forming a new variable scope for its properties. The class follows the same schema as the outer workflow class. It features another workflow method (`run()` in this case) and three activity methods, most notably the `receiveResponse` method, where the client checks its memory for messages arriving at the specified message group channel. In accordance with the loop-condition of the original subprocess, the content of the workflow method is executed in an infinite loop. The subprocess itself is run in a thread, which will eventually be interrupted by the event handler thread, thus breaking out of the loop.



```
  public class Client_requestTaxi extends AbstractWorkflowBean {
     protected TaxiResponse globalBest;
     protected Boolean noResponse;
     protected String appID;
     protected String location;
     protected String destination;

     public final static String ACTION_WORKFLOW = "requestTaxi";

     public void doStart() throws Exception {
        super.doStart();
        join("TaxiResponseTo"+appID+"");
     }

1    @Expose(name=ACTION_WORKFLOW, scope=ActionScope.GLOBAL)
     public Serializable[] workflow(String currentLoc, String currentDest) {
        location = currentLoc;
        destination = currentDest;
2       sendRequest();
3       Thread waitForReplies = new Thread(new WaitForReplies_Sub());
4       EventHandler wait30Seconds_timeout =
              new TimeoutEventHandler(waitForReplies, 30000);
        try {
           waitForReplies.start();
           wait30Seconds_timeout.start();
           waitForReplies.join();
           wait30Seconds_timeout.stopEventHandler();
        } catch (InterruptedException e) {
        }
5       if (noResponse) {
8          String taxiID = "none available";
           return new Serializable[] {taxiID};
        } else {
6          notifyTaxis();
7          String taxiID = globalBest.getTaxiID();
           return new Serializable[] {taxiID};
        }
     }

     // continued on right side
```

```
2    protected void sendRequest() {
        TaxiRequest request = new TaxiRequest(location, appID, destination);
        send("TaxiRequest", request);
     }

6    protected void notifyTaxis() {
        SelectedTaxi selectedTaxi = new SelectedTaxi(globalBest.getTaxiID());
        send("Notification", selectedTaxi);
     }

3    class WaitForReplies_Sub implements Runnable {
        protected TaxiResponse currentResponse;

        public void run() {
           noResponse = true;
3.0        while (true && ! checkInterrupted()) {
3.1           receiveResponse();
3.2           if (currentResponse.compareTo(globalBest) > 0) {
3.4              updateGlobalBest();
              } else {
3.3              doNothing();
              }
           }
        }

3.1     protected void receiveResponse() {
           TaxiResponse response = (TaxiResponse)
                 receive("TaxiResponse", TaxiResponse.class, -1);
           noResponse = false;
           currentResponse = response;
        }

3.3     protected void doNothing() {
        }

3.4     protected void updateGlobalBest() {
           globalBest = currentResponse;
        }
     }
  }
```

Fig. 4. Example: Taxi Request Service. Corresponding parts in the process diagram and the code are numbered correspondingly.

The Agent Bean for the *Taxi* role is similarly structured, and thus is not shown here. The main difference is that its workflow method is not exposed as an action, but is invoked by a memory observer listening for the request messages sent by the client role. The observer is attached to the agent's memory in the `doStart()` method (one of the life-cycle methods, which is started when the agent is started). The workflow method itself is rather straightforward, with an if-statement representing the first gateway, and an event-handler for the second. The logic for the *evaluate request* task can either be provided via the task's *script* attribute, or it can be implemented in the generated Java code.

6.1 Discussion

Using the domain-specific scripting language JADL, agent behaviours can be expressed in a very compact and readable way, but the overall expressiveness (e.g., the supported event types) is limited by the scripting language. JIAC Agent Beans, on the other hand, have the full expressiveness of the Java language at their disposal. Thus, basically everything that can be modelled in a BPMN diagram can be mapped to an Agent Bean.

While the resulting workflow method for complex processes can become somewhat bulky – particularly if event handling is used – its structured form as well as the separation into workflow methods and activity methods keeps the resulting code reasonably clear. Like in WADE, individual activity methods can be altered or extended without risk of losing the changes after the code is generated anew. The reason why this is important is that while BPMN is well suited for high-level behaviour, graphically modelling low-level algorithms and such would be too laborious. This way, those can be added to the generated code.

One potential problem might be raised by the extensive use of Java threads for event handling. We are currently investigating ways of integrating the event handling into the agent's main thread. Another alternative would be to move away from the current workflow methods towards a more interpreter-like approach, memorizing the current state of the process and executing one activity method in each step of the agent's execution cycle. Particularly for long-running processes this might be beneficial.

Regarding the high expressiveness of the generated Agent Beans and the good performance of compiled Java code when compared to the interpreted JADL scripts, the mapping from BPMN to JIAC Agent Beans is suited best for modelling and generating core components of the multi-agent system, while the mapping to JADL is of much use for creating dynamic behaviours and services to be deployed and changed at runtime.

7 Conclusion

In this paper, we have presented an approach for creating multi-agent systems from process models, combining the mapping from BPMN to JADL [18] with ideas borrowed from WADE [10]. The result is a transformation from BPMN process diagrams to JIAC Agent Beans, generating one method for the workflow as a whole, and one method for each individual activity. The resulting Agent Bean classes are highly expressive and at the same time well structured and readable. Being based on the wide-spread Business Process Model and Notation, the process diagrams are easy to understand and the mapping also supports important aspects such as communication and interaction and event handling, which are particularly suited for being modelled visually.

Comparing our approach with related works, our impression is that using a powerful yet high-level notation like BPMN provides for more expressive agent behaviours, in particular w.r.t. communication and event handling. On the other hand, we acknowledge that a simpler notation that is more streamlined to the requirements of agent engineering may be easier to learn, somewhat balancing the benefit of using an established industry standard.

Of course, it depends on the application to be developed whether process modelling in general and BPMN in particular are appropriate ways for designing the system: Particularly when intensive communication and event handling is involved, graphical process modelling notations have their benefits, but visually depicting every detail of a complex algorithm can become rather cumbersome.

Our work has not yet reached the maturity of some of the related approaches. Still, using the mapping proposed and exemplified in this paper, it is possible to model complex and distributed multi-agent systems by means of BPMN and to generate readily executable agent behaviours from the process diagrams. Also, while we decided to use JIAC in this work, the bulk of the mapping could be applied to other agent frameworks, as well.

7.1 Future Work

While the mapping can already be used for generating useful agent behaviours, it is not yet completed. First, there are still aspects of BPMN that are not covered by the mapping, such as some of the less common event types. Second, there are aspects of agents that can not yet be modelled adequately with BPMN.

One such issue that we want to tackle in the future is the modelling of goals and other kinds of dynamic behaviour by means of BPMN. Without those, the resulting agent systems, strictly following the process diagram, are rather procedural and inflexible. One promising approach is to use the *ad-hoc* subprocess for this task, executing a certain set of activities in no predefined order until a given *completion condition* is met. However, this is still work in progress.

Complementary to the transformation to JIAC code, we are currently working on a process interpreter agent bean. Similar to the JADL interpreter agent, this will allow to pass processes to the agent at runtime and to have that agent execute one or more of the roles in that process [29]. Without the additional layer of abstraction of the scripting language, this approach is expected to have the same expressive power as the generated JIAC bean while at the same time being more dynamic. Also, this will allow for monitoring and visualizing the current state of the running process by linking the process interpreting agent to the modelling tool.

The downside of the interpreter approach is that the entire behaviour has to be modelled in the process diagram or has to be made available as callable services, since there is no possibility to manually extend the generated code. Thus, we see the upcoming interpreter as a way to dynamically deploy very high-level processes to the running agent, while the core behaviours of the agent would still be created in a combination of process modelling, code generation, and manually extending and refining the generated code.

References

1. Baldoni, M., Baroglio, C., Capuzzimati, F.: 2COMM: A commitment-based MAS architecture. In: Cossentino, M., El Fallah Seghrouchni, A., Winikoff, M. (eds.) EMAS 2013. LNCS (LNAI), vol. 8245, pp. 38–57. Springer, Heidelberg (2013)
2. Bauer, B., Müller, J.P., Odell, J.: Agent UML: A formalism for specifying multiagent software systems. In: Ciancarini, P., Wooldridge, M.J. (eds.) AOSE 2000. LNCS, vol. 1957, pp. 91–103. Springer, Heidelberg (2001)

3. Bellifemine, F., Poggi, A., Rimassa, G.: JADE – a FIPA-compliant agent framework. Internal technical report, Telecom Italia (1999), part of this report has been also published in Proceedings of PAAM 1999, London, pp. 97–108 (April 1999)
4. Bergenti, F., Caire, G., Gotta, D.: Interactive workflows with WADE. In: 2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, pp. 10–15 (2012)
5. Bergenti, F., Caire, G., Gotta, D., Long, D., Sacchi, G.: Enacting BPM-oriented workflows with Wade. In: Proceedings of the 12th Workshop on Objects and Agents, Rende, CS, Italy, pp. 112–116 (July 2011)
6. Braubach, L., Pokahr, A., Jander, K., Lamersdorf, W., Burmeister, B.: Go4Flex: Goal-oriented process modelling. In: Essaaidi, M., Malgeri, M., Badica, C. (eds.) Intelligent Distributed Computing IV. SCI, vol. 315, pp. 77–87. Springer, Heidelberg (2010)
7. Burmeister, B., Arnold, M., Copaciu, F., Rimassa, G.: BDI-agents for agile goal-oriented business processes. In: Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008), pp. 37–44. International Foundation for Autonomous Agents and Multiagent Systems, Richland (2008)
8. Cabri, G., Puviani, M., Quitadamo, R.: Connecting methodologies and infrastructures in the development of agent systems. In: Proceedings of the International Multiconference on Computer Science and Information Technology (IMCSIT 2008), pp. 17–23. IEEE, Wisla (2008)
9. Caire, G.: WADE User Guide, Version 2.6. Telecom Italia (July 2010), `http://jade.tilab.com/wade/doc/WADE-User-Guide.pdf`
10. Caire, G., Gotta, D., Banzi, M.: WADE: A software platform to develop mission critical applications exploiting agents and workflows. In: Berger, M., Burg, B., Nishiyama, S. (eds.) Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008) – Industry and Applications Track, pp. 29–36 (May 2008)
11. Caire, G., Quarantotto, E., Porta, M., Sacchi, G.: WOLF – An Eclipse plug-in for WADE. In: Proceedings of the ACEC 2008 (2008)
12. Calisti, M., Greenwood, D.: Goal-oriented autonomic process modeling and execution for next generation networks. In: van der Meer, S., Burgess, M., Denazis, S. (eds.) MACE 2008. LNCS, vol. 5276, pp. 38–49. Springer, Heidelberg (2008)
13. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Information & Software Technology 50(12), 1281–1294 (2008)
14. Endert, H., Küster, T., Hirsch, B., Albayrak, S.: Mapping BPMN to agents: An analysis. In: Baldoni, M., Baroglio, C., Mascardi, V. (eds.) Agents, Web-Services, and Ontologies Integrated Methodologies (AWESOME), pp. 43–58 (2007)
15. Hirsch, B., Konnerth, T., Burkhardt, M., Albayrak, S.: Programming service oriented agents. In: Calisti, M., Dignum, F.P., Kowalczyk, R., Leymann, F., Unland, R. (eds.) Service-Oriented Architecture and (Multi-)Agent Systems Technology. Dagstuhl Seminar Proceedings, vol. 10021, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany (2010)
16. JIAC Development Team: JIAC – Java Intelligent Agent Componentware, Version 5.1.5. DAI-Labor, TU Berlin (February 2014), `http://www.jiac.de`
17. Küster, T., Heßler, A.: Towards transformations from BPMN to heterogeneous systems. In: Ardagna, D., Mecella, M., Yang, J. (eds.) BPM 2008 Workshops. LNBIP, vol. 17, pp. 200–211. Springer, Heidelberg (2009)
18. Küster, T., Lützenberger, M., Heßler, A., Hirsch, B.: Integrating process modelling into multi-agent system engineering. Multiagent and Grid Systems 8(1), 105–124 (2012)

19. Lützenberger, M., et al.: A multi-agent approach to professional software engineering. In: Winikoff, M., El Fallah Seghrouchni, A., Winikoff, M. (eds.) EMAS 2013. LNCS (LNAI), vol. 8245, pp. 156–175. Springer, Heidelberg (2013)
20. Mendling, J., Lassen, K.B., Zdun, U.: Transformation strategies between blockoriented and graph-oriented process modelling languages (2005)
21. Muehlen, M.z., Recker, J.: How much language is enough? Theoretical and practical use of the business process modeling notation. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 465–479. Springer, Heidelberg (2008)
22. Object Management Group: Business process model and notation (BPMN) version 2.0. Specification formal/2011-01-03, Object Management Group (August 2011)
23. Ouyang, C., Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M., Mendling, J.: From business process models to process-oriented software systems. ACM Transactions on Software Engineering and Methodology 19(1), 1–37 (2009)
24. Pokahr, A., Braubach, L.: Reusable interaction protocols for workflows. In: Workshop on Protocol Based Modelling of Business Interactions (2010)
25. Pokahr, A., Braubach, L., Jander, K.: Unifying agent and component concepts – Jadex active components. In: Dix, J., Witteveen, C. (eds.) MATES 2010. LNCS (LNAI), vol. 6251, pp. 100–112. Springer, Heidelberg (2010)
26. Recker, J.C.: BPMN modeling – who, where, how and why. BPTrends 5(3), 1–8 (2008)
27. Sierra, C., Rodríguez-Aguilar, J.A., Blanco-Vigil, P.N., Arcos-Rosell, J.L., Esteva-Vivancos, M.: Engineering multi-agent systems as electronic institutions. UPGRADE: European Journal for Informatics Professional V (4), 33–39 (2004)
28. Tan, P.S.: Automated Generation of JIAC AgentBeans from BPMN Diagrams. Diploma thesis, Technische Universität Berlin (November 2011)
29. Voß, M.: Orchestrating Multi-Agent Systems with BPMN by Implementing a Process Executing JIAC Agent Using the Visual Service Design Tool. Master thesis, Humboldt Universität Berlin, realized with support of DAI-Labor, TU Berlin (May 2014)
30. Yu, E.S.: Social modeling and *i**. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 99–121. Springer, Heidelberg (2009)