Mapping BPMN to Agents: An Analysis

Holger Endert, Tobias Küster,
Benjamin Hirsch, and Sahin Albayrak

DAI-Labor, Technische Universität Berlin
Faculty of Electrical Engineering
and Computer Science
{holger.endert|tobias.kuester|
benjamin.hirsch|sahin.albayrak}@dai-labor.de

Abstract. In industry the development of software applications is usually a complex and demanding task, and the design and the technical realisation is often spread among different roles, which leads to a time consuming and error-prone exchange of knowledge. In order to ensure the correct translation from business idea to implementation it is crucial to allow for the correct and complete exchange of information between these roles.
In this paper, we describe an automated mapping from business process diagrams to agent concepts that simplify the transfer of knowledge between the roles involved in the software development process. Our approach benefits from building upon an intuitive visual specification language on the one hand, and from using a powerful and flexible execution platform on the other.

1 Motivation

Multi-agent systems (MAS) arguably provide an answer to the creation of complex distributed applications which are manageable and adapt to the state of the environment. However, even though agent research has been ongoing for more than a decade now, it still is a mainly academic subject. The rapid uptake of technologies such as web-services, which aim squarely at the same problem space as multi-agent systems [9], however suggests that a corresponding demand is existent in industry. We believe that one important reason for the slow adoption of agents in industry is the disconnect that exists between business- and multi-agent oriented software development.

It appears that while introducing mentalistic notions to model agents is a very intuitive approach, business users tend to think in terms of processes, and business entities – and they are the ones that make overall design decisions! Therefore, we approach the issue of designing multi-agent frameworks from the vantage point of the business user, and provide a mapping from the Business Process Modeling Notation (BPMN [12]), a graphical language used to represent business processes, to agent concepts.

Choosing the BPMN as the source language for our mapping, and thus as the language for designing distributed business applications, our approach can

offer certain advantages, such as providing a simple and intuitive graphical notation. Although its basic concepts are simple, further specification options make the language sufficiently expressive. The BPMN is also suitable for defining processes of different levels of abstraction. As the focus lies on the process model of an application, an extension will be required in order to integrate the support for structured data types. It seems reasonable to rely on existing specification languages for that purpose. Since the BPMN has emerged from a standardisation effort for business processes by the OMG, it will most likely play a role in software specification in the near future, which additionally encourages our approach.

Using multi-agent systems as target model provides the capabilities of a powerful execution environment and an intuitive abstraction model. For instance, having an explicit representation of a process participant, i.e. an agent, makes the application more accurate w.r.t. its structure. In contrast to that, a mapping to BPEL (e.g. as presented in [12] or [19]) would discard this structural information completely. MAS offer some further properties, for instance their intuitive design through mentalistic notions, scalability and flexibility, which help to deploy the resulting applications. Because of the variety of existing multi-agent frameworks, we have to define the requirements for the target model as generically as possible. Therefore we decided to map into BDI-type MAS, because these seem to be most suitable to capture the process model of a business process diagram.

To summarise, our main contribution of this paper is to provide one step towards the connection of business process design and the design of multi-agent systems by means of an automated mapping. Our focus lies on the identification of BDI-related concepts that can represent specific elements of the BPMN, rather than specifying all details of the control flow, which is already covered by several other authors (see for example [13] or [19]). The key feature of this approach is to facilitate the correct and fast transformation of the original concepts into code that can be used directly or with adaptations within the resulting business application.

The rest of this paper is structured as follows. In Section 2 and 3 we will provide a short introduction to BPMN and agents and state a formal description for both. In Section 4 we will describe the actual mapping from BPMN to agents. Then we will present related work in Section 5, and finally we will conclude and give rise to future work in Section 6.

2 BPMN

The *Business Process Modeling Notation* (BPMN), which is maintained by the Object Management Group, is a graphical notation for describing various kinds of processes. The main notational elements in BPMN are *FlowObjects*, that are contained in *Pools* and connected via *Sequence-* and *MessageFlows*. They subdivide in *Events*, atomic and composite *Activities* and *Gateways* for forking and joining. SequenceFlows describe the sequence in which the several FlowObjects have to be completed, while MessageFlows describe the exchange of messages

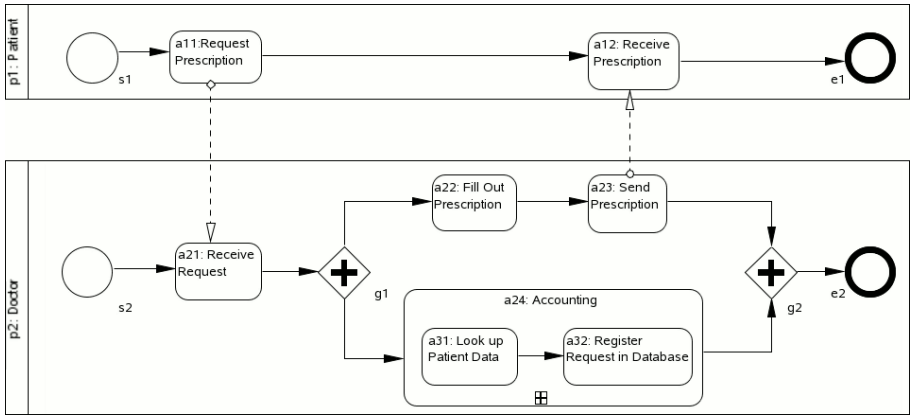

Fig. 1. BPMN example diagram.

between Pools. Thus, BPMN combines the definition of local workflows and the interaction between them.

An example diagram is shown in Figure 1. As can be seen, the meaning of the diagram can be understood intuitively and without any specific knowledge about the semantics of BPMN. This strength is at the same time a weakness in that BPMN is more of a graphical notation than a formal one, which is required for automatic interpretation. Although the specification includes some semantics and even defines a mapping to WS-BPEL [12, Chapter 11], much of the semantics is especially tailored for this mapping and the focus of the specification is clearly the visual representation.

2.1 Graph-Based BPMN Representation

For the purpose of our mapping, we can simplify the BPMN by discarding all layout information. What is left is a graph-like structure with several types of vertices and edges that should satisfy certain properties in order to be a correct business process diagram (BPD). Therefore, it is straightforward to define a BPD in terms of a graph and graph properties[1]. This representation has the additional advantage of allowing further analyses that help to implement correct diagrams. In general, a BPD is defined as the following graph structure:

Definition 1. *(BPD-Graph) - Let* $BPD = (O, F, src, tar)$ *be a graph with*

- O *— the set of nodes (objects) in the BPD-Graph.*
- F *— the set of edges (message and sequence flows) in the BPD-Graph.*
- $src, tar : F \rightarrow O$ *two functions, which identify the source and target objects of each edge.*

[1] More specifically, *typed attributed graphs* are required in order to capture the attributes of the nodes that are necessary for the mapping.

In order to distinguish the different nodes and edges in a BPD-Graph, some additional notations are required. Therefore, let O be partitioned into the disjoint subsets O^E, O^A, O^G, O^P, where

- O^E — the set of event-nodes, which can be further partitioned into the disjoint subsets O^E_S, O^E_E, O^E_I, i.e. start-, end- and intermediate events.
- O^A — the set of activity-nodes, which can be further partitioned into the disjoint subsets O^A_{At}, O^A_{Sub}, i.e. the atomic activity nodes and the subprocess nodes.
- O^G — the set of gateway-nodes, which can be further partitioned into the disjoint subsets O^G_S and O^G_M, i.e. the splitting and the merging gateway nodes. These can again be partitioned into the subsets for exclusive (XOR), inclusive (OR) and parallel (AND) split and merge gateways ($O^G_{S,X}, O^G_{S,O}, O^G_{S,A}, O^G_{M,X}$, $O^G_{M,O}$ and $O^G_{M,A}$).
- O^P — the set of pool-nodes.

Elements of the BPD have attributes, which are also relevant for the mapping, because they contain the data and the parameters used within the process. In slight abuse of notion, we refer to attributes of an element by using the common dot-syntax. For example, if p is a pool, the term $p.name$ refers the *name*-attribute of this element.

Note that not every BPD graph, as presented here, refers to a valid diagram according to the specification. Hence, a BPD is said to be *correct* if it satisfies a set of additional properties. Actually, these properties claim even more than correctness, namely that the diagram is *normalised*, i.e. it conforms to a canonical representation. BPDs can be normalised through a graph transformation, such that our approach is not generally limited to a subset of diagrams. Normalised BPDs simplify the mapping because there are less cases to consider. Both the normalisation and correctness is analysed in [10], and hence not part of this paper.

In the next section, we will present a formal description of the agent concepts, which we are going to use in the mapping.

3 Agents

Throughout the literature, there is no single and universally accepted definition of the term agent (for a number of possible definitions see e.g. [11]). Nevertheless, certain properties are required in order to translate a complete BPD graph. In our opinion, agents that follow the BDI paradigm [6] are best suited to capture all the functionality that is expressed with BPMN. Hence, we propose to use agents that are capable of performing BDI-related reasoning on goals, plans, intentions and beliefs. Subsequently, we define an agent as a tuple containing plans, goals, intentions, beliefs and an identifier:

Definition 2. *(Agent) - An agent is a tuple Λ = (id, Π, Γ, I, Θ) with*

- *id — A unique string that allows to identify an agent.*

– *Π — A set of plans,that the agents knows.*
– *Γ — A set of goals an agent is trying to achieve.*
– *I — A set of intentions (i.e. selected plans).*
– *Θ — A set of beliefs an agent knows.*

Plans define an agents possible behaviour. A plan may be selected for execution by the agent, which results in an intention. Goals refer to usual achievement goals, and lead to new intentions in order to fulfil them. Beliefs is the set of facts that an agent assumes to be true. These building-blocks are subsequently specified in more depth.

3.1 Plans

Plans define the actions of an agent that may be carried out in order to achieve a certain goal. A plan must have a defined signature and an executable (or interpretable) script that is capable of performing BDI-related operations, such as manipulating the beliefs or goals of an agent. Additionally, it must allow to organise these elements using control flow, e.g. for conditional or parallel branching. Often plans consist also of preconditions and effects, but since we do not require them in this work, we omitted them for simplicity reasons.

Definition 3. *(Plan) - A plan is a tuple π = (Name, In, Out, X), where the elements are defined as follows:*

– *Name — The name that identifies the plan.*
– *In — The list of input variables of the plan.*
– *Out — The list of output variables of the plan.*
– *X — The script of the plan, which is a sequence of control flow elements together with operations for manipulating the agents state.*

Variables occurring in the In- and Out-lists are defined as tuples, containing a name and a type $\vartheta := (Name, Type)$. The script X must support at least the following operations:

– **invoke(***name***)** — Invokes a plan for execution.
– **addGoal(γ)** — Adds a goal to the goal-base.
– **send(μ)** — Sends a message μ.
– **receive(μ)** — Receives a message μ.

3.2 Goals

A goal in this work corresponds to a usual *achievement goal*, and is related to a plan. This relation is defined via the plans and the goals signatures. Both must fit to each other by having the same name, and the *In*- and *Out*-lists must match to each other w.r.t. the sizes and types of the elements.

Definition 4. *(Goal) - A goal is a tuple γ = (Name, In, Out), where the name is an identifier, and In and Out are lists of data-elements.*

Note, that the in- and output elements are not restricted to variables here, but can contain any data elements.

3.3 Data and Beliefs

Data can be any (evaluated) expression and may be bound to a variable. A belief (fact) is data, which is located in the belief-base of an agent, and is referable via the name.

Definition 5. *(Fact) - A fact is a tuple θ = (Name, Value, Type). The Name is a unique identifier, the Value represents the content of the fact, and the Type belongs to a formal definition of a domain, e.g. to an ontology.*

As can be seen, adding a variable to the belief-base results in a fact.

3.4 Messages

The exchange of messages is one major aspect of business processes. For the purpose of the mapping, we have to deal with messages in multiple cases (send-tasks, events, etc.). We therefore summarize our minimum requirements on messages here.

Definition 6. *(Message) - A message is a tuple μ = (Name, Sender, Receiver, Content), with:*

- *Name — The id of the message.*
- *Sender — The unique id of the sender (agent).*
- *Receiver — The unique id of the receiver (agent).*
- *Content — A list of data.*

Although agents (and multi-agent systems) often contain many more aspects, no further elements are necessary for the mapping we will describe in the next section.

4 Mapping

The mapping we present is similar to that defined in the BPMN specification, which uses WS-BPEL [8] as target model. Since we use agents instead, the results will have several (desired) differences and advantages. For instance, agent frameworks that conform to the FIPA agent platform specification [1] provide infrastructure services that enable agents to find and cooperate with each other in a flexible manner. Thus, an application is easier to use and distribute among execution platforms. Another interesting point is, that agents can be modelled in terms of mental attributes, such as goals, beliefs and intentions, and hence their implementation can be quite intuitive and on a higher level of abstraction.

The approach we take is the definition of a graph transformation system, which is specified by a set of transformation rules, that map elements of the source model (BPMN) to elements of the target model (agents). The mapping is separated into a set of different tasks which have to be completed in a specific order. First of all, for any BPD, we apply a normalization process, that translates a

given graph into its canonical representation. Thereafter we check for syntactical inconsistencies by evaluating graph properties. Then we analyze the semantics of the BPD using an approach based on petri nets, as presented in [10]. Finally, if everything went well so far, the mapping itself is executed. In this work, we only deal with the final part, the mapping, focusing on the most relevant BPMN elements and their counterparts in the agents model (Subsection 4.1). There, all rules are in the form

$$LHS \Longrightarrow RHS,$$

where LHS (left-hand side) is a pattern that is searched in the source graph, and RHS (right-hand side) contains the elements that are added to the target model. The rules are not fully specified in that they do not provide $NACs$ (negative application conditions) or a reference model. The former is in most cases used to ensure that each element is mapped only once. The latter stores the progress of the mapping and may contain additional temporary elements, that simplify the mapping.

4.1 Mapping of Nodes

We initially start with providing the mapping-rules for nodes in a top to bottom manner. The most top-level nodes of a BPD are pools, representing the participants within a diagram. These are mapped directly to agents by applying this rule:

Rule 1 *(Pools) - Let $p \in O^P$ be a pool of a given BPD.*

$$p \Longrightarrow \Lambda := (p.name, \emptyset, \emptyset, \emptyset, \emptyset)$$

So far, an agent does not possess any specific knowledge in terms of plans, goals, intentions or beliefs. These elements are added by the application of subsequent rules. For each pool, there exists exactly one process that is translated into a plan and added to the agents plan-library.

Rule 2 *(Process) - Let p be a pool, and Λ be the agent, that was created from p. Let further be x the process of p, i.e. $p.process = x$.*

$$x \Longrightarrow \pi := (x.name, [], [], [])$$
$$\Pi_\Lambda := \Pi_\Lambda \cup \{\pi\}$$

The plan is given with a name, an empty in- and output-list, and an empty script, which has to be filled during the mapping of its control flow. The in- and output-lists depend on the start- and end-events of the process. If these are message-events, the message-properties are used to create the parameters and the results as follows:

Rule 3 *(Start-Event) - Let $e \in O_S^E$ be a start-event with $e.trigger = Message$. Let further π be the plan, that was created from the process, in which e is located.*

$$\forall p \in e.message.properties \Longrightarrow \vartheta := (p.name, p.type)$$
$$In_\pi := append(In_\pi, \vartheta)$$

Rule 4 *(End-Event) - Let $e \in O_E^E$ be an end-event with $e.result = Message$. Let further π be the plan, that was created from the process, in which e is located.*

$$\forall p \in e.message.properties \Longrightarrow \vartheta := (p.name, p.type)$$
$$Out_\pi := append(Out_\pi, \vartheta)$$

Start events of the type *Timer* and *Rule* may also be used in another context. Both should result in reactive behaviour, where the former requires that agents are aware of time, and the latter that they can monitor their beliefs w.r.t. certain conditions. Other end event-types refer to control flow, such as the termination or failure of a process, and are not detailed here.

Sub-processes are used to create hierarchical and reusable structures within a BPD. Either they refer to completely independent processes (*independent sub-process*), which may be defined outside the given BPD, or they can be included into a parent process (*embedded sub-process*). The *independent sub-process* is mapped onto a goal, revealing one major strength of multi-agent systems. In this case, the corresponding plan can be provided by any agent, if the target platform supports *Yellow Pages Services*. The *embedded sub-processes* is mapped onto a plan, because it contains an own workflow. Additionally an operation for invoking that plan is created. Note, that the *invoke*- and *addGoal*-operations have to be added into the correct place within the script. This is done during the control flow mapping, and hence they are not added into any script by the given rules:

Rule 5 *(Embedded Sub-Process Activity) - Let $x \in O_{Sub}^A$ be an activity, with $x.subProcessType = Embedded$. Let further be Λ the agent, that was created from the pool, in which x is located.*

$$x \Longrightarrow \pi := (x.name, [], [], [])$$
$$\Pi_\Lambda := \Pi_\Lambda \cup \{\pi\};$$
$$\textbf{invoke}(\textbf{x.name})$$

Rule 6 *(Independent Sub-Process Activity) - Let $x \in O_{Sub}^A$ be an activity, with $x.subProcessType = Independent$.*

$$x \Longrightarrow \gamma := (x.processRef.name, [], [])$$
$$\textbf{addGoal}(\gamma)$$
$$\forall i \in x.inputPropertyMaps \Longrightarrow In_\gamma := append(In_\gamma, eval(i))$$
$$\forall o \in x.outputPropertyMaps \Longrightarrow Out_\gamma := append(Out_\gamma, eval(o))$$

Note that the elements of the property-maps, which are used to pass the data to the goal in Rule 6, are string expressions. Hence we cannot be more specific than interpreting the string, which is done using the *eval* function. Therefore it must conform to the syntax that is used to refer to a data-element in the script language of the target platform.

Communication in BPMN is specified by using *send* or *receive* tasks. These are simply mapped onto asynchronous speechacts. The rule for the send task is given exemplarily. The corresponding rule for the receive task is defined analogously. Only the *taskType* and the created operation is different.

Rule 7 *(Send-Task) - Let $x \in O^A_{At}$ be an activity with $x.taskType = Send$. Let further be μ the message, that should be sent (and is mapped from the message-attribute of x).*

$$x \Longrightarrow \mathbf{send}(\mu)$$

We note, that there are some nodes left that are not covered so far. We will not provide rules for each of them here, but discuss their mapping briefly. First of all, a very helpful BPMN node is the script-task, which may contain arbitrary code in its *script*-attribute. With this, it is easy to define functionality, that cannot be expressed otherwise. The mapping of reference nodes (task or sub-process of type *reference*) depends on the nodes they refer to. Some other elements are not supported yet, for instance the mapping of transactions, which is also an open issue in the mapping to WS-BPEL.

The next subsection describes, how these elements interact with each other by means of modifying and passing data.

4.2 Mapping of Data Flow

BPMN is not intended to model data [12, p. 34], and thus has to rely on other sources. Additionally, since the main purpose of BPMN is the visualisation of the work flow, data handling is not very comfortable. In most cases, it is completely hidden, and can only be specified within non-visible attributes of the nodes.

Data flow is addressed by assigning values to properties that can be attributed to every activity (task or sub-process) in a BPD. Since each relevant aspect of a property (i.e. its type, name or value) has to be given as string, the designer is free to choose any representation, and can simply adopt the specific language of the target model. The general approach for mapping the data handling of activities is given in Figure 2. In addition to the *operation* (which may be for instance an *invoke*), its local variables have to be created (as facts), and values have to be assigned to them. Assignments after the execution of the operation can be used to bind the results to globally accessible facts again for further processing. Since we utilise the belief-base of an agent for storing and accessing data, which in contrast to properties of activities is always globally accessible, the *cleanup* section removes them from the belief-base afterwards.

Some specific elements also possess messages (send- and receive tasks, events of type message), which also provide properties for capturing data. Messages are

```
[define facts]
[assignments]
<operation>
[assignments]
[cleanup facts]
```

Fig. 2. General data handling for activities

mapped onto their counterparts in the agent model, and in the case of the tasks are inserted before the speechact operation into the script, which passes them as parameter (see Rule 7). The assignments again assure that sent and received data is bound to the corresponding facts/variables.

The most specific mapping of data handling is given for goals, that result from *independent sub-processes* (see Rule 6). There, data is passed using the attributes *input-* and *outputPropertyMap*, which are sets of expressions. These map the properties between the two processes, and are used for this purpose in the agents model as well. Note, that the expressions are again represented as ordinary strings, and hence have to be encoded in the target language already.

The next subsection presents some issues concerning the mapping of the control flow.

4.3 Mapping of Control Flow

The mapping of control flow will be applied after the several BPMN elements have been mapped to equivalent elements of the agent description language. The purpose of the mapping of control flow is to arrange this collection of atomic elements into structures such as sequences, if-else blocks or loops. These structures then make up a plan's script. The mapping is quite intuitive to understand and at the same time complicated to realize and highly dependent on the targeted agent language's capabilities.

Basically, the flow objects contained in one pool will map to one plan. Sequence flows determine the order of execution. Gateways define the extend of structured elements, such as conditional blocks, parallel blocks or loops, depending on the gateway's type (AND, XOR, ...) and the number of incoming and outgoing sequence flows. In the case of BPMN also event handlers (intermediate events on an activity's boundary) have to be taken into account, which can complicate the resulting structured workflow by some amount, making it necessary to skip parts of the workflow in case an event handler is triggered.

Well-structured workflows can be mapped in quite a simple rule-based bottom-up approach. A set of rules is used to identify the few basic structures that make up a complex workflow – sequences, blocks, loops, etc. – and combine the target elements that have been mapped from the involved source elements to structures accordingly. After that, either the source model itself or a reference model, connecting the source model with the target model, has to be reduced by replacing

the successfully mapped structure with an atomic marker-node, referencing the new structure in the target model, so that a rule can not be applied twice for the same element. This process is repeated until the model can not be reduced any further.

However, in some cases, when the BPMN workflow is not well-structured, it is hard to map the control flow that is defined within BPMN to any kind of script, because the languages have a different power of expressiveness. Usually, graph-like languages such as BPMN allow to specify control flow, which is generally not reproduceable in block-oriented languages.

Some examples of such are workflows containing an AND-split gateway being followed by a OR-join gateway, resulting in a lack of synchronization and multiple instances of the workflow after the joining gateway, which can not be expressed easily in block-structured languages. Other examples are all kinds of overlapping blocks and loops and interconnected parallel workflows.

Some of these problems can be tackled by duplicating parts of the workflows, by spawning child-processes or by introducing auxiliary variables. Obviously, it is highly favorable if this is done programmatically and the user does not have to consider these workarounds.

These problems have been discussed in a number of papers and will not be reconsidered here. For further information, please refer to [13, 15, 19, 21].

4.4 Mapping-Example

In this section we will illustrate the mapping using the simple business process diagram introduced in Figure 1. In the course of this example we will use the identifiers found on each of the diagram nodes to refer to the nodes.

Besides the visual nodes the BPD also needs some non-visual attributes, such as properties and assignments, which will be given in Table 1. In the following we will use a notation like $prop = (name : type)$ for properties and $assign = (to \leftarrow from, assignTime)$ for assignments. Note also that in Figure 1 $a11.message$ is equal to $a21.message$ and $a12.message$ is equal to $a23.message$.

Element	**Properties**	**Assignments**
p1.process	$req : String, ans : String$	$req \leftarrow$ "$need\_meds \ldots$", $before$
a11.message	$msg\_req : String$	
a11		$msg\_req \leftarrow req, before$
a12		$ans \leftarrow msg\_ans, after$
p2.process	$req : String, ans : String$	
a21		$req \leftarrow msg\_req, after$
a22		$ans \leftarrow$ "$recipe\_for\_meds \ldots$", $after$
a23.message	$msg\_ans : String$	
a23		$msg\_ans \leftarrow ans, before$

Table 1. The example diagram's non-visual attributes

Firstly, both the Patient pool ($p1$) and the Doctor pool ($p2$) are mapped to agents (Rule 1), each holding a plan for the pool's process (Rule 2).

$$p1 \Longrightarrow \Lambda_{p1} := (\text{“}Patient\text{”}, \{\pi_{p1}\}, \emptyset, \emptyset, \emptyset)$$
$$\pi_{p1} := (\text{“}proc\_patient\text{”}, [\,], [\,], [\,])$$

The plan's *In*- and *Out*-lists are empty, since the start- and end-events in this simple example diagram are not of type *Message* and thus do not have a mapping. Also, the activities $a22$, $a31$ and $a32$, which are not further specified (e.g. as script-tasks), will simply be referred to by their *id*s.

The mapping of the first send-task would look like the following (Rule 7):

$$a11.message \Longrightarrow \mu_{req} := (\text{“}msg\_1\text{”}, \text{“}Patient\text{”}, \text{“}Doctor\text{”}, [msg\_req])$$
$$a11.assignments \Longrightarrow \mathbf{ass_{a11}}$$
$$a11 \Longrightarrow \mathbf{send}(\mu_{\mathbf{req}})$$

Since it will be up to the actual agent system how to realise the assignments between the global properties and the message properties we will leave this open and refer to the mapped assignments of activity x as $\mathbf{ass_x}$. The other send- and receive-tasks will map to similar structures and will not be explicitly stated here.

The mapping of the subprocess $a24$ would result in the following (Rule 5):

$$a24 \Longrightarrow \pi_{a24} := (\text{“}accounting\text{”}, [\,], [\,], [\,])$$
$$\Pi_{p1} := \Pi_{p1} \cup \{\pi_{a24}\}$$
$$\mathbf{invoke}(\mathbf{accounting})$$

Finally the control flow has to be mapped, which can be done using a set of rules like those described in Subsection 4.3. The results of the mapping can be found in Table 2: Each pool has been mapped to one agent – Λ_{p1} and Λ_{p2} – holding a plan for the pool's process and its subprocesses.

The properties of the process, i.e. the global variables, can be found in the agent's fact base Θ. The script elements that have been created from the various flow objects have been arranged in sequences and blocks and inserted into the plans' scripts[2]. In the next section we will have a look on other work that has been done in this area so far.

5 Related Work

Considering our longterm-goal, which is to use an intuitive graphical process notation for designing multi-agent systems, few related work exists. On the workflow level, the usage of petri nets and extensions (e.g. CPN) is studied for a

[2] We use the notation $x_1; x_2; x_3$ for sequences of script elements and $x_1 \parallel x_2$ for parallel execution, which are the only control structures needed in this example.

$$
\begin{array}{|l|}
\hline
\Lambda_{p1} = (\text{“}Patient\text{”}, \Pi_{p1}, \emptyset, \emptyset, \Theta_{p1}) \\
\Pi_{p1} = \{\pi_{p1}\} \\
\Theta_{p1} = \{req, ans\} \\
\pi_{p1} = (\text{“}proc\_patient\text{”}, [\,], [\,], X_{p1}) \\
X_{p1} = [\mathbf{ass_{p1.proc}}; \mathbf{ass_{a11}}; \mathbf{send}(\mu_{\mathbf{req}}); \mathbf{receive}(\mu_{\mathbf{ans}}); \mathbf{ass_{a12}}] \\
\mu_{req} = (\text{“}msg\_1\text{”}, \text{“}Patient\text{”}, \text{“}Doctor\text{”}, [msg\_req]) \\
\hline
\Lambda_{p2} = (\text{“}Doctor\text{”}, \Pi_{p2}, \emptyset, \emptyset, \Theta_{p2}) \\
\Pi_{p2} = \{\pi_{p2}, \pi_{a24}\} \\
\Theta_{p2} = \{req, ans\} \\
\pi_{p2} = (\text{“}proc\_doctor\text{”}, [\,], [\,], X_{p2}) \\
X_{p2} = [\ \mathbf{receive}(\mu_{\mathbf{req}}); \mathbf{ass_{a21}}\ ; \\
\qquad (\mathbf{a22}; \mathbf{ass_{a22}}; \mathbf{ass_{a23}}; \mathbf{send}(\mu_{\mathbf{ans}})) \parallel \mathbf{invoke}(\mathbf{accounting})\ ] \\
\pi_{a24} = (\text{“}accounting\text{”}, [\,], [\,], [\mathbf{a31}; \mathbf{a32}]) \\
\mu_{ans} = (\text{“}msg\_2\text{”}, \text{“}Doctor\text{”}, \text{“}Patient\text{”}, [msg\_ans]) \\
\hline
\end{array}
$$

Table 2. Results of the mapping

while, for example by Aalst et al. [2]. In [18], petri nets are even directly used to model multi-agent systems. Our approach differs from them in that it uses a notation that is more common to business users, and thus increases the accessibility to industry. As shown in [10], a semantical analysis is possible as well by translating business process diagrams into petri nets, but that can be done without any knowledge of the user about petri nets.

Our approach also relates to model driven architectures (MDA) in the general case, and in case for multi-agent systems, as for instance described in [5]. Again, the specification languages used there are usually used by software designers rather than by business users, and therefore are not suited to bridge the disconnect between industry and agent-oriented software design. On the other hand, they provide a set of interesting methodologies, and are more complete than our current work, and are thus a good reference point for further investigations.

As the importance and impact of SOA and related technologies starts to become clear in the area of agent technologies, a number of people have worked on different ways of incorporating knowledge and experiences of the different fields. Therefore we want to mention some work that is at least partially related to what we have presented in this paper. Casella et al. [7] and Mantell [16] have worked on creating tools to translate UML diagrams to BPEL. Mantell translates UML activity diagrams to executable BPEL code, while Casella et al. start from protocol diagrams designed in the UML extension Agent-UML [3] and create abstract BPEL processes. In addition to that, there exists work that incorporates web services into multi-agent systems, which in combination with the previously mentioned approaches would have a similar effect as our work, but is less straightforward. For instance, Bozzo et al. [4] apply the BDI paradigm in order to create adaptive systems based on webservices. Here, they start from a BDI-type multi-agent system (based on AgentSpeak [20]) and extend it to use web services as primitive actions. A similar work was developed by Vidal et

al. [23]. Walton [24] suggests to differentiate between agent interaction protocols and agent body, and therefore to allow web services (bodies) to be seamlessly incorporated into multi-agent system, and vice versa. Finally, for an overview on existing approaches regarding the benefits of combining web services and agency, see [9].

6 Conclusion

In this paper, we have proposed a way to close the gap that exists between the different roles which participate in the development of business applications, by defining a mapping from BPMN to agent concepts. Following this approach, we defined a set of (abstract) rules, showing how business processes can be modelled in terms of BDI-type agents. We also argued that the mapping is only one aspect out of a set of tasks that can be supported by a tool. We additionally proposed the usage of syntactical and semantical verification, combined with a normalisation of BPDs. In future, this may lead to a complete methodology for designing and implementing multi-agent systems. One aspect that is still missing is the data handling, which has to be incorporated in an appropriate manner.

6.1 Implementation

A transformation from BPMN to JIAC IV (Java-based Intelligent Agent Componentware) [22], a multi-agent framework based on the BDI paradigm, has lately been developed in the course of a diploma thesis [14].

For the purpose of this transformation a BPMN editor has been implemented using Eclipse GMF. It can be used to create and validate BPDs and to initiate the transformation. Since there is no XML schema or similar given for BPMN, the editor's domain model had to be created from scratch and thus is not compatible with other BPMN editors. However, it supports each single attribute given in the BPMN specification. The mapping has been implemented as a transformation tool using both a top-down pass through the BPMN model and a rule-based transformation, which can be subdivided in four stages: Normalization, element mapping, structure mapping and clean-up.

Although the mapping from BPMN to JIAC is not yet fully specified and there is still some work to do to support the transformation of unstructured workflows, the basics are working fine and simple BPMN diagrams can be transformed to JIAC multi-agent systems.

6.2 Future Work

After having defined a set of basic rules for a mapping from BPMN to agents there is still much work to be done in the future. First of all, we want to explore the mapping of the work flow as completely as possible. Therefore we will also identify any further requirements that are needed on the agents side in order to

capture the desired functionality. To this end, it seems to be necessary to define or use an existing platform independent agent metamodel.

On a higher level, we plan to extend our approach to a complete methodology. The first thing to consider is the integration of data types as a fixed part of the design process. It is planned to combine BPMN with OWL [17] and OWL-S respectively, such that the extension is supported by well-founded concepts. An additional advantage is that this allows to define semantical services in terms of preconditions and effects.

Another interesting research task concerns the mapping into the opposite direction, i.e. from agents to BPMN. Since multi-agent systems are naturally complex, a good visualisation of their workflow, the interaction protocols and organisational structures would increase the understanding of existing systems significantly. Ideally, this would also help to identify in which parts the two models diverge in the power of expressiveness, and what extensions are needed in order to adjust this discrepancy.

References

1. Foundation for Intelligent Physical Agents (FIPA). Specification index, 2007.
2. W. Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
3. B. Bauer, J. P. Müller, and J. Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering, 1^{st} International Workshop, AOSE 2000, Revised Papers*, volume 1957 of *LNCS*, pages 91–104. Springer-Verlag, 2001.
4. L. Bozzo, V. Mascardi, D. Ancona, and P. Busetta. COOWS: Adaptive BDI Agents meet Service-Oriented Computing – extended abstract. In M. P. Gleizes, G. A. Kaminka, A. Nowé, S. Ossowski, K. Tuyls, and K. Verbeeck, editors, *Proceedings of the 3^{rd} European Workshop on Multi-Agent Systems (EUMAS'05)*, pages 473–484. Koninklijke Vlaamse Academie van Belie voor Wetenschappen en Kunsten, 2005.
5. A. BRANDO, V. SILVA, and C. LUCENA. A model driven approach to develop multi-agent systems. Technical report, Departmento de Informtica - Pontifcia Universidade Catlica do Rio de Janeiro - PUC-Rio, 2005.
6. M. E. Bratman. *Intentions, Plans, and Practical Reason*. Havard University Press, Cambridge, MA, 1987.
7. G. Casella and V. Mascardi. From AUML to WS-BPEL. Technical Report DISI-TR-06-01, Dipartimento di Informatica e Scienze dell'Informatione, Università di Genova, 2006.
8. O. Committee. Web Services Business Process Execution Language (WS-BPEL) Version 2.0. Technical report, Oasis, 2007.
9. I. Dickinson and M. Wooldridge. Agents are not (just) web services: considering BDI agents and web services. In *Proceedings of the 2005 Workshop on Service-Oriented Computing and Agent-Based Engineering (SOCABE'2005), Utrecht, The Netherlands*, July 2005.
10. H. Endert, B. Hirsch, T. Küster, and S. Albayrak. Towards a Mapping From BPMN to Agents. In J. Huang, R. Kowalczyk, Z. Maamar, D. Martin, I. Müller, S. Stoutenburg, and K. Sycara, editors, *Service-Oriented Computing: Agents, Semantics, and Engineering (SOCASE-2007)*, volume 4504. Springer Verlag, May 2007.

11. S. Franklin and A. Graesser. Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents. In *ECAI '96: Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, pages 21–35, London, UK, 1997. Springer-Verlag.
12. O. M. Group. Business Process Modeling Notation (BPMN) Specification. Final Adopted Specification dtc/06-02-01, OMG, 2006. `http://www.bpmn.org/Documents/OMGFinalAdoptedBPMN1-0Spec06-02-01.pdf`.
13. B. Kiepuszewski, A. H. M. ter Hofstede, and C. Bussler. On Structured Workflow Modelling. In *CAiSE '00: Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, pages 431–445, London, UK, 2000. Springer-Verlag.
14. T. Küster. Development of a Visual Service Design Tool providing a mapping from BPMN to JIAC. Master's thesis, Technische Universität Berlin, 2007.
15. R. Liu and A. Kumar. An Analysis and Taxonomy of Unstructured Workflows. In W. M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Business Process Management*, volume 3649, pages 268–284, 2005.
16. K. Mantell. From UML to BPEL — Model Driven Architecture in a Web Services World. Technical report, IBM, 2005. `http://www-128.ibm.com/developerworks/webservices/library/ws-uml2bpel/`.
17. D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language. W3C Recommendation, 2004. `http://www.w3.org/TR/owl-features/`.
18. D. Moldt and F. Wienberg. Multi-Agent-Systems Based on Coloured Petri Nets. In *ICATPN*, pages 82–101, 1997.
19. C. Ouyang, W. van der Aalst, M. Dumas, and A. ter Hofstede. Translating BPMN to BPEL. Technical Report BPM-06-02, BPMCenter.org, 2006.
20. A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In R. van Hoe, editor, *Agents Breaking Away, 7^{th} European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'96*, volume 1038 of *LNCS*, pages 42–55, Eindhoven, The Netherlands, January 1996. Springer Verlag.
21. W. Sadiq and M. E. Orlowska. Analyzing process models using graph reduction techniques. *Inf. Syst.*, 25(2):117–134, 2000.
22. R. Sesseler. *Eine modulare Architektur für dienstbasierte Interaktionen zwischen Agenten*. PhD thesis, Technische Universität Berlin, 2002.
23. J. M. Vidal, P. Buhler, and C. Stahl. Multiagent Systems with Workflows. *IEEE Internet Computing*, 8(1):76–82, January/February 2004.
24. C. Walton. Uniting Agents and Web Services. In *Agentlink News*, volume 18, pages 26–28. AgentLink, 2005.