Behavior Alignment as a Mechanism for Interaction Belief Matching

Gerben G. Meyer

September 11, 2006

Abstract

When two or more agents have to perform an interaction that is part of a business process instance (BPI) they can either follow an established protocol, or use their own experience and judgment. In the second case, there is no protocol enacted, or if it is, it does not cover special circumstances that occur in this particular instance. In this case, the agents have to use their own experience, acquired in previous and similar kind of interactions. Before the interaction, the agents will build an intended behavior (their own course of action) and also will presume what the other agents are doing (expected behavior). The intended behaviors of the agents interacting are not always matching, in this case the interaction will not be completed successfully. The process of collaboratively changing behavior for successful matching is called alignment.

In this thesis, a formalism will be presented to define agents' behaviors (as exhibited in agent to agent interactions), by an extension of Petri Nets. Secondly, it will be shown how behaviors of different agents can be aligned using specific alignment policies. A mechanism using a neural network is proposed for automatic choosing of an alignment policy by the agent. Furthermore, a method how agents can overcome their lack of experience by escaping is described. This research will make the system more reliable, and to reduce the necessary human intervention. Possible future directions of research are also pointed out.

Title:
Behavior Alignment as a Mechanism for Interaction Belief Matching

Keywords:
agent behavior modeling, alignment policies, automatic behavior alignment, interactions, escape mode

Supervisors:
Nick B. Szirbik
Gerard R. Renardel de Lavalette

"Every single man or woman who has stood their ground, everyone who has fought an agent has died. But where they have failed, you will succeed." - Morpheus (The Matrix)

Preface

In front of you lies the master thesis, which is written for my graduation project of the master program Computer Science (variant Intelligent Systems) at the University of Groningen. It is the result of the 7 months of research I have done at The Agent Laboratory, which is part of the Faculty of Management and Organization of the same university.

In this thesis, a new and formal way how agents' behaviors can be modeled is proposed and explained. This way of modeling is particularly focused on agent interactions. Furthermore, a technique is proposed how agents can automatically change their behavior when it seems to be inappropriate for a certain interaction. As far as we know this is the first attempt to apply Petri Net techniques in adaptive agent behavior.

This thesis is not the only result of my research done at The Agent Laboratory during this period. Software has been developed to simulate agents, in order to see if the way behavior is modeled is appropriate, and if agents really can handle an interaction when their behavior seems to be insufficient. Also two accepted papers have been written. The first paper has been accepted for the student session of *The 8th European Agent Systems Summer School (EASSS 2006)*, and the second paper has been accepted for *The 3rd Workshop on Anticipatory Behavior in Adaptive Learning Systems (ABiALS 2006).* Both papers can be found in the appendices.

With this thesis, I finish my graduation project. First, I would like to thank the people with who I worked in the lab, Nick Szirbik, Gijs Roest and Marco Stuit, for the good discussions and the great social environment. Special thanks goes to Gijs Roest for his assistance with developing the software for this project. I thank my supervisors Nick Szirbik and Gerard Renardel de Lavalette for their help and feedback during this project. Finally, I would like to thank my parents for their support and making it possible for me to attend university.

Gerben G. Meyer
Groningen, The Netherlands
September 2006

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In The Agent Laboratory (TAL) [29] research is focused on modeling, simulating and deploying agents to support business processes. Furthermore, TAL is investigating novel architectures for building Multi-Agent Systems. The modeling language TALL (The Agent Lab Language) is used for modeling business processes with agents. This language is originally proposed by De Snoo [26], but is recently extended by Stuit [27]. For simulating and deploying, the AGE (Agent Growing Environment) framework is currently developed [24].

1.1 The need for alignment

When two or more agents have to perform an interaction that is part of a business process instance (BPI) they can either follow an established protocol, or use their own experience and judgment.

In the first case, the agents should learn or be instructed about the protocol beforehand. They should be able to perform all the tasks that are imposed by the use of the protocol (in other words: have the necessary skills to play that particular role in that particular interaction). The protocol should be clear, captured in some (semi)formal way on paper and easy to explain to everybody that has the chance to use it. The protocol should be based on previous experience, success rates, quality standards, etc.

In the second case, there is no protocol enacted, or if it is, it does not cover special circumstances that occur in this particular instance. In this case, the agents have to use their own experience, acquired in previous similar kinds of interaction. The agents will build before the interaction an *intended behavior* (their own course of action) and also will presume what the other agents are doing (*expected behavior*). These beliefs will form together a mental state what we call the *intended interaction* or *interaction belief* of the agent in that situation (before the interaction is started).

Imagine that there are two agents ready to start the interaction. They can proceed immediately or they can exchange the intended behavior. If the interaction is between two "dumb" software agents, it is possible that the two behaviors will result in an interaction that will block, and it will not achieve its goal.

For two humans, the deadlock can be solved somehow. If both agents realize after the interaction started that there is a mismatch in their beliefs about how the interaction will go, they can roll-back and change in a collaborative way their behavior in a way that promises success in ending the whole interaction. In successive steps of trial and error, they usually manage to finish the interaction. If exactly the same agents are interacting again in the future, they can use the final behavior (which they memorize) that led to success. Of course, this will not insure success again, because the circumstances (the environment factors) can be different.

The process of collaboratively changing behaviors will be called *alignment*.

1.2 The alignment process

Typically, between humans, this alignment happens on the fly, during the interaction. The behavior the agent has presumed of the other agent (expected behavior) changes by the messages received of the other agent, if the messages are different than expected. The agent can try to adapt his own behavior, so the interaction can still succeed, despite the different messages. This way of alignment can be in most situations very effective. However, because the interaction can still fail, the trial and error takes time and effort, and when a final solution is built, it can be one which is very poor in terms of efficiency and/or quality.

Two or more behaviors that have been aligned successfully are named *matching behaviors*. The behaviors defined by a protocol have to be always matching.

If the agents do not manage to align their behavior on the fly, their complete behaviors have to be compared. Because agents do not know each other behaviors, an agent standing above them who can access both behaviors is the most suitable to align these behaviors. This agent can also be a software agent, but can also be human. This is called *escape* mode, if one of the agents is calling for a higher level agent to align the behaviors. A higher level agent can also *intervene* in the interaction process a priori or during the interaction, to align the agents intended behaviors. In both cases, the Deus ex machina adapts the behavior of the agents [24].

A third possibility which also happens in real life could be to align their behavior a priori, by exchanging each own local behavior and analyze and discuss beforehand. Analysis can reveal potential deadlock and mismatches. The interacting agents can align their behaviors before the interaction starts. Of course, this will not ensure success, but it provides the interaction instance with a matching set of behaviors. But for agents (in the AGE-framework) this can be a difficult approach. Two agents could have a meta-level interaction before the real interaction about this real interaction, and try to align their behaviors a priori the real interaction in this way. But there is no way to ensure this meta-level interaction is aligned, and that this interaction will reach its goal of two aligned behaviors for the real interaction. It could work if a protocol is defined for this task, but of course this still will not ensure success. Without a protocol, the best negotiator will probably get what he wants.

If we consider simulation, on-the-fly alignment can be used for a priori alignment. The agents can simulate first a trial and error alignment session, and come with a solution that is satisfactory in terms of matching and performance.

So the total problem of alignment for this thesis boils down to two problems:

- (automatically or manually) alignment of the agents own intended behavior on the fly.

- (automatically or manually) alignment of intended behaviors of multiple agents by an agent at a higher level (e.g. Deus ex machina).

A third possibility would be to align the intended behaviors by the agents themselves a priori the interaction. This happens in real life, but is probably to difficult for this thesis.

1.3 Alignment as a way to match behaviors

In organizations where protocols and previous experience is scarce, the agents who are interacting within the business processes are forced to do alignment all the time. If experience is building up, the agents will exhibit more and more often matching behavior, making the business processes to run more smoothly. Also, the business processes will settle themselves in patterns that can be easily identifiable from a central perspective and make them explicitly supported by workflow enactment systems. Finally, we may end with a global protocol imposed by the workflow description. We have poured the concrete over our organization and this is now solid. This solution is useful for very strict organizations, like the military, the tax-office, safety critical environments (railways control), etc.

However, environment, agent structure, organizational structure (roles), are continuously changing. That will change behaviors and process structure, leading to necessary alignments. Most of the business organizations encounter high rate of change. For these organizations, workflow and protocol based solutions are not effective. They have to find ways to make alignment natural, easy and efficient.

Most important, we should always assume that intended behaviors are never matched. Each interaction instance (i.e. business process) is unique, and any previous experience, however extensive, will not ensure success. Room and means for alignment should be integral part of any agent system that serves a highly changing environment. Even if protocols are enacted, it is not sure that they can assure success in unforeseen circumstances.

1.4 Related work

This research is based on two different research fields. On one hand, it is based on workflow research, which applies Petri Nets for modeling business processes (See for example [8]). The research of Van der Aalst on this subject also investigates modeling inter-organizational workflows [1], which is close to the concept of agent interaction. However, this approach is based on a central perspective, where the behaviors of the two organizations is modeled from a central point of view. He also describes when an inter-organizational workflow is sound [2, 3, 5]. This can however only be done when there is a overall perspective.

On the other hand, this research is based on agents. When two agents are communicating, the global behavior is not central represented, because the agents are executing their behavior distributed. They however have an (implicit) representation of the expected behavior of the other agent, on which her own behavior is based. This approach is inspired by anticipatory systems by Ekdahl [11]:

> "Anticipation means that a system is able to make predictions about what will happen when faced with a special situation and act in accordance with this prediction. This implies that an anticipatory systems will take into consideration future possible events in deciding what to do in the present situation. Thus an effect is not explained completely by a cause but by expectations. (...) More sophisticated anticipatory systems are those which also contain its own model, are able to change model and to maintain several models, which imply that such systems are able to make hypotheses and also that they can comprehend what is good and bad."

Looking at this description of anticipatory systems, our agents can be called anticipatory. The intended behavior of an agent is based on the expected behavior (which is the future behavior) of the agent interacting with. Furthermore, an agent can maintain several models of her own behavior, and is able to change its own model if needed. This is actually one of the main subjects of this master thesis.

As far as we know, this is the first approach where research of those two fields are combined, where workflow techniques are used for modeling and verifying agent behavior.

1.5 Structure of report

This report is structured as follows. Section 2 will discuss Behavior Nets, as an extension of Petri Nets, which will be used for modeling agent behaviors. In chapter 3 the three layer approach of behaviors will be explained. Chapter 4 operations are introduced on how Behavior Nets can be modified. Chapter 5 will discuss the interaction concept, Interaction Belief cycle and examples of not aligned interactions are given. Chapter 6 and 7 will give methods for aligning two different behaviors, from a local and a global perspective. This reports ends with a description of the created software in section 8 and a conclusion and discussion in section 9 and 10.

Chapter 2

Behavior Nets, an extension of Petri Nets

Petri Nets are a class of modeling tools, which originate from the work of Petri [21]. Petri Nets have a well defined mathematical foundation, but also a well understandable graphical notation [25]. Because of the graphical notation, Petri Nets are powerful design tools, which can be used for communication between the people who are engaged in the design process. On the other hand, because of the mathematical foundation, mathematical models of the behavior of the system can be set up. The mathematical formalism also allows validation of the Petri Net by various analysis techniques.

In this chapter, the definition of the classical Petri Nets will first be discussed. Secondly, the workflow nets by Van der Aalst will be discussed, including some properties and analysis techniques proposed by him. This chapter will end with the definition of Behavior Nets, which will be used throughout this paper to model agents behavior.

2.1 Classical Petri Nets

The classical Petri Net is a bipartite graph, with two kind of nodes, *places* and *transitions*, and connections between these nodes called *arcs*. A connection between two nodes of the same type is not allowed.

Definition of Petri Net A Petri Net is a tuple $PN = (P, T, F, M_0)$

- P is a finite set of places

- T is a finite set of transitions

- $F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs

- $M_0 : P \to \mathbb{N}$ is the initial marking

A place p is an input place for a transition t if there is a directed arc from p to t. A place p is an output place for a transition t if there is a directed arc from t to p. The notation $\bullet t$ is used to denote the set of all input places for transition t, and $t\bullet$ is the set of all output places for this transition. The same can be done for places, $p\bullet$ is the set of all transitions which have p as input, $\bullet p$ is the set of all transitions having p as output.

A marking $M = (M(p_1), M(p_2), \ldots, M(p_{|P|}))$ represents a *state* of the modeled system, which is the distribution of tokens over the set of places. Starting from an initial marking M_0 a new marking M is *reachable* if it can be reached by means of a change to the state of the system. A transition t is *enabled*, if every input node contains at least one token, so $\forall p \in \bullet t : M(p) \geq 1$. An enabled transition may fire, which will change the current marking M_1 into a new marking M_2. The effect of firing a transition t can be expressed as $M_1 \xrightarrow{t} M_2$. Firing a transition t will *consume* one token from each of its input places, and *produce* one token in each of its output places. In this thesis, I will use the term *marking* and *state* interchangeably.

In the remainder of this section, I will give the definitions of some properties of Petri Nets [5].

Live A Petri Net PN with marking M is live, iff for every reachable state M_1 and every transition t there is a state M_2 reachable from M_1 which enables t.

Bounded A Petri Net PN with marking M is bounded iff for every reachable state and every place p the number of tokens in p is bounded.

Strongly connected A Petri Net is strongly connected iff for every pair of nodes x and y, there is a directed path leading from x to y.

Free-choice A Petri Net is a free-choice Petri Net iff for every two places p_1 and p_2 either $(p_1 \bullet \cap p_2 \bullet) = \emptyset$ or $p_1 \bullet = p_2 \bullet$.

2.2 Workflow nets

Workflow nets are a specific subclass of Petri Nets. The objective of a workflow net is the modeling of how a specific case must be processed. The definition of a workflow net is the following [3, 5]:

Definition of workflow net A Petri Net $PN = (P, T, F, M_0)$ is a WF-net (Workflow net) if and only iff:

- There is one source place $i \in P$ such that $\bullet i = \emptyset$.

- There is one sink place $o \in P$ such that $o\bullet = \emptyset$.

- Every node $x \in P \cup T$ is on a path from i to o.

A workflow handles *cases*. A WF-net has one input and one output place, because any case handled by the WF-net is created when the process starts, and deleted when the process is completely handled, i.e., the WF-net specifies the life cycle of the case. The third requirement is to avoid dangling tasks and conditions, i.e., tasks and conditions which do not contribute to the processing of cases. For a more extensive discussion about WF-nets, the reader is referred to [3, 5].

One of the most important properties for WF-nets given by Van der Aalst is soundness [3]. Note that symbol i is used to denote both place i and the state with only one token in place i.

Definition of soundness A WF-net $PN = (P, T, F, M_0)$ is sound if and only if:

- For every state M reachable from state i, there exists a firing sequence leading from state M to state o. Formally:

$$\forall_M (i \rightarrow^* M) \Rightarrow (M \rightarrow^* o)$$

- State o is the only reachable state from state i with at least one token in place o. Formally:

$$\forall_M (i \rightarrow^* M \wedge M \geq o) \Rightarrow (M = o)$$

- There are no dead transitions in PN with state i. Formally:

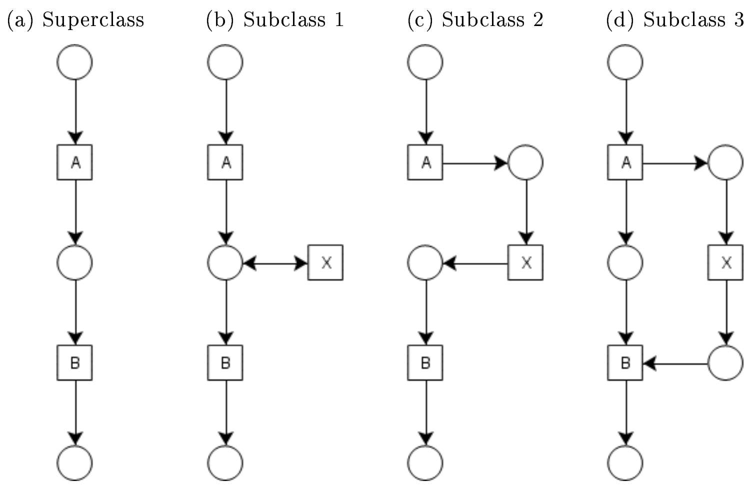
$$\forall_{t \in T} \exists_{M_1, M_2} i \rightarrow^* M_1 \rightarrow^t M_2$$

The first requirement states that starting from the initial state i, it is always possible to reach the state with one token in place o. The second requirement states that the moment a token is put in place o, all the other places should be empty. Also the term proper termination can be used to describe the first two requirements. The last requirement states that there are no dead transitions in the initial state i.

Inheritance Inheritance is well defined for objects in Object-Oriented programming languages. One class is a subclass of another class if and only if it can do what the other classes can do. Moreover, it will typically add new functionality. But for workflows and behaviors modeled as Petri Nets, this is not so straightforward. For this problem, Van der Aalst has identified four different notions of inheritance [6]: *protocol inheritance*, *projection inheritance*, *protocol/projection inheritance* and *life-cycle inheritance*. The two important for this thesis are discussed next.

Projection inheritance is based on abstraction. If it is not possible to distinguish x and y when arbitrary tasks of x are executed, but when only the effects of tasks that are also present in y are considered, then x is a subclass of y with respect to projection inheritance. In other words, when every task in x which

Figure 2.1: Examples of projection inheritance



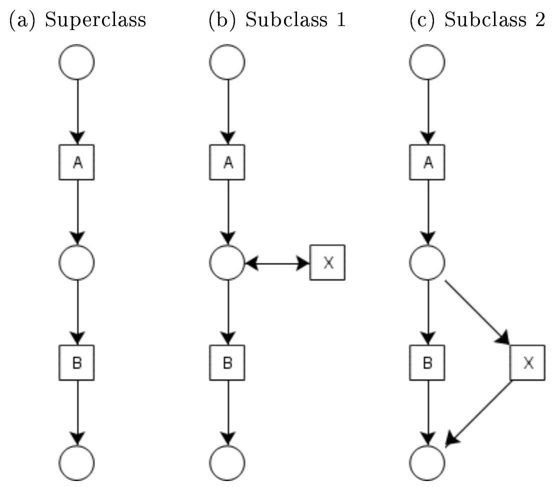
is not in y is replaced by an "empty" and not observable task, and afterward x and y are not distinguishable, then x is a subclass of y.

Examples of projection inheritance are shown in figure 2.1. Workflow (b), (c) and (d) are all three subclasses of workflow (a) with respect to projection inheritance. When hiding task X in (b), the workflow is equivalent to (a). The same is true for (b), if the detour of task X is hidden, the observable behavior is the same as (a). If in (d) the parallel execution of X is hidden, this workflow is also equivalent to (a), and thus a subclass. Note that none of the subclasses shown in figure 2.2 on the next page are subclasses according to the notion of projection inheritance, except (b). Workflow (c) of that figure is not a subclass of (a) by projection inheritance, hiding task X would give the possibility to skip task B, and would thus yield in a different behavior.

Protocol inheritance is based on encapsulation. If it is not possible to distinguish x and y when only tasks of x that are also present in y are executed, then x is a subclass of y. In other words, if all tasks in x which are not in y are blocked, and thus not executed, then when x and y are not distinguishable, x is a subclass of y.

Examples of protocol inheritance are shown in figure 2.2 on page 18. Workflow (b) and (c) are both subclasses of workflow (a) with respect to protocol inheritance. In both examples, blocking task X yields into the same workflow as (a). Note that none of the subclasses shown in figure 2.1 are subclasses according to the notion of protocol inheritance, except (b). Workflow (c) and (d) of that figure are not a subclass of (a) by protocol inheritance, because both workflows would deadlock when blocking task X.

Figure 2.2: Examples of protocol inheritance



2.3 Colored Petri Nets

In classical Petri Nets, there is only one type of tokens, the “black” indistinguishable tokens. In contrast to that, in Colored Petri Nets (or CP-nets), there are different type of tokens, and thus can tokens be distinguished from each other, hence they are called colored. A token in a CP-net can have all kind of variables attached to it, numbers, strings, lists, etc..

Because of the variables of the tokens, guards can be added to the net, to ensure a token has certain content, otherwise a transition may not consume the token. Furthermore, CP-nets have expressions and bindings, which can alter the content of the tokens. More information about Colored Petri Nets can be found in [18].

2.4 Behavior Nets

In the following, I give the formal definition of Behavior Nets, which is partially based on and an extension of Workflow nets, Self-Adaptive Recovery Nets (see [16]) and Colored Petri Nets. An example of such a Behavior Net can be seen in figure 2.3 on page 20.

Definition of Behavior Net A Behavior Net is a tuple $BN = (\Sigma, P, Pm, T, Fi, Fo, i, o, L, D, G, B)$ where:

- Σ is a set of data types, also called color sets

- P is a finite set of places

- Pm is a finite set of message places
- T is a finite set of transitions (such that $P \cap Pm = P \cap T = Pm \cap T = \emptyset$)
- $Fi \subseteq ((P \cup Pm) \times T)$ is a finite set of directed incoming arcs
- $Fo \subseteq (T \times (P \cup Pm))$ is a finite set of directed outgoing arcs such that:

$$\forall p \in Pm : \bullet p = \emptyset \vee p\bullet = \emptyset$$

- i is the input place of the behavior with $\bullet i = \emptyset$ and $i \in P$
- o is the output place of the behavior with $o\bullet = \emptyset$ and $o \in P$
- $L : (P \cup Pm \cup T) \to A$ is the labeling function where A is a set of labels
- $D : Pm \to \Sigma$ denotes which data type the message place may contain
- G is a guard function which is defined from Fi into expressions which must evaluate to a Boolean value such that:

$$\forall f \in Fi : [Type(G(f)) = bool \wedge Type(Var(G(f))) \subseteq \Sigma]$$

- B is a binding function defined from T into a set of bindings b, which binds values (or colors) to the variables of the tokens such that:

$$\forall t \in T : [Type(B(t)) \in \Sigma \wedge \forall v \in Var(B(t)) : [b(v) \in Type(v)]]$$

The set of types Σ defines the data types tokens can be, and which can be used in guard and binding functions. A data type can be arbitrarily complex, it can be for example a string or a integer, but it can also be a list of integers, or combinations of variable types.

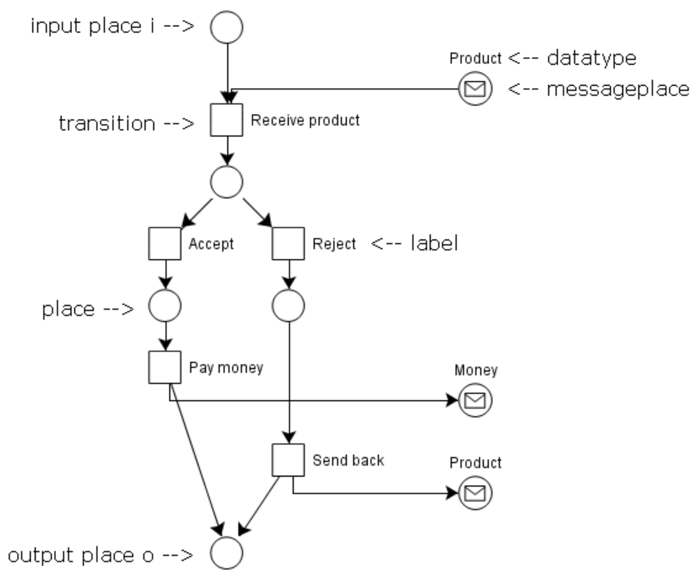
The places P and Pm and the transitions T are the nodes of the Behavior Net. All these three sets should be finite. The extension of classical Petri Nets is the addition of the set Pm which are nodes for sending and receiving messages during an interaction. Such a message place is either a place for receiving or for sending messages, it cannot be both.

Fi and Fo are sets of directed arcs, connecting the nodes. An arc can only be from a place to a transition, or from a transition to a place. By requiring the sets of arcs to be finite, technical problems are avoided, such as the possibility of having a infinite number of arcs between two nodes.

Execution a behavior is part of an interaction process, an instance of the behavior is created when the interaction starts, and deleted when the interaction is completed. For this reason, the Behavior Net also has to have one input and one output node, like with WF-nets.

With function L, a label can be assigned to every node. This has no mathematical or formal purpose, but makes the Behavior Net better understandable in the graphical representation.

Figure 2.3: Example of a Behavior Net



Function D denotes what data type a message place may contain. This is useful for determining on which message place an incoming message will be placed. Because the two (or more) behaviors in an interaction are distributively executed, message places of both behaviors cannot be connected directly with each other, the behaviors do not have to be aligned. However, TALL uses currently interaction belief diagrams where message places are connected to transitions of the expected behavior. In this thesis, the supplementary information of the expected behavior is not used.

Function G is the guard function, which expresses what the content of a token has to be, to let the transition consume the token from the place. Function G is only defined for Fi, because it makes no sense to put constraints on outgoing edges of transitions.

Transitions can change the content of a token. Binding function B defines per transition, what the content of the tokens produced by the transition will be. Bindings are often written as for example: $(T1, < x = p, i = 2 >)$, which means that transition $T1$ will bind value p to x and value 2 to i. The values assigned to the variables of the token (which data type must be in Σ) can be constants, but can also be values of the incoming token, or values from the knowledge or belief base of the agent.

Visualization Figure 2.3 shows how a Behavior Net can be graphically notated. This figure shows most the the Behavior Net constructs. The figure illustrates the behavior of a buyer, who will receive a product, and either accepts the product and pays the money, or reject the product and sends the it back.

Verification Verification of Behavior Nets can be done using workflow verification techniques of Van der Aalst [5, 3]. First, it can be checked if there is only one input place and only one output place. Secondly, the soundness of the behavior can be checked. For this purpose, the message places Pm have to be removed from the behavior, and the functions G and B can be ignored. Because only the behavior of one agent without the interaction components is verified, this is referred to as *local soundness*. When the Behavior Nets of the agents participating in the interaction are combined into one Petri Net, i.e. the message places of the different behaviors are mapped on each other, the *global soundness* of the interaction can be verified. More details about local and global soundness can be found in [1].

Chapter 3

Three layered behavior models

As a modeled behavior cannot be executed immediately, a three layer approach is proposed in this chapter. These three layers, and the relations between them are shown in figure 3.1. The behaviors in all three layers are based on Petri Net extensions. This three-layer approach is based on the play-in/play-out approach of Harel and Marelly [17]. When the behavior is executed in the compiled layer, as discussed later on in this chapter, it is still possible to see the progress of this execution in the TALL-layer, which is used for the modeling of the behavior.

3.1 TALL-layer

The top layer of the three layers is called the TALL-layer. The TALL-layer is the layer used by humans to model the agent's behavior, and is the visualization used when the behavior is executed, i.e. when running an interaction. For this layer a reduced version of the process diagrams of the modeling language TALL (The Agent Lab Language) is used. All model constructs which can be used are shown in figure 3.2 on page 23 and discussed below.

Place A place in TALL is just like a place in Petri Nets, it is a state where a token can be.

Start place A start place is just like a place, but it typically has no incoming edges from other nodes.

End place A end place is just like a place, but it typically has no outgoing edges to other nodes.

Figure 3.1: Three layers behavior

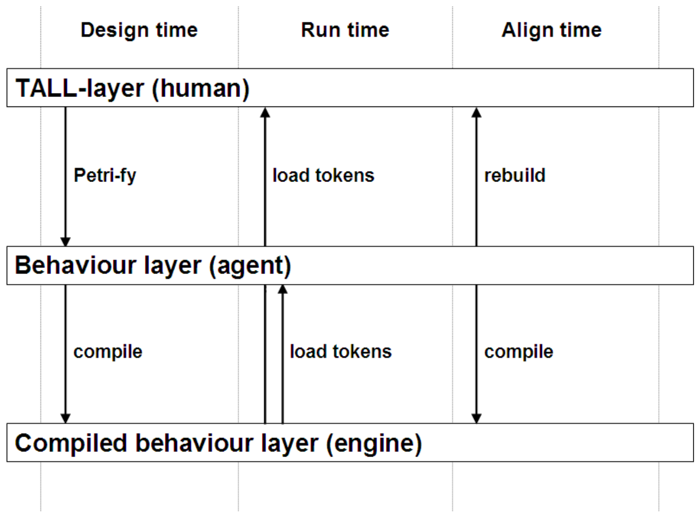


Figure 3.2: Model constructs for TALL-layer

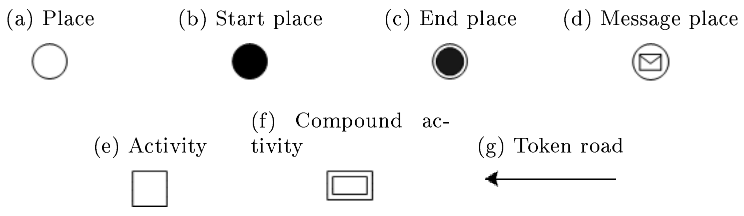
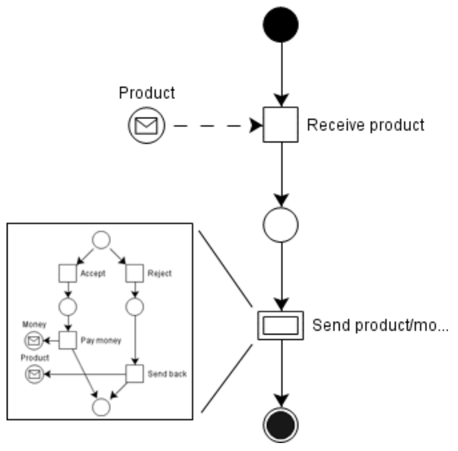


Figure 3.3: TALL-layer example



Message place A message place is used for sending and receiving messages to other agents in the interaction. It always is a virtual place; it is not part of the behavior of the agent itself, it only states that a message should be sent or received.

Activity An activity is always connected with one or more places by incoming edges, and one or more places by outgoing edges. It is just like an transition in Petri Nets, and it can have a set of actions, which it executes when consuming the tokens from the incoming edges. Such an action can for example be changing the content of the token.

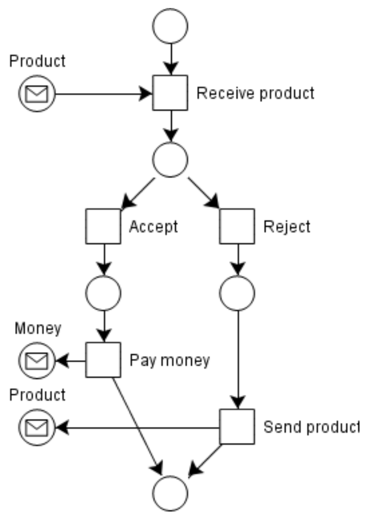
Compound activity A compound activity is a TALL process diagram in itself. In this way, a multi-level behavior can be modeled, to abstract away from certain details. The diagram in the compound activity should have as many start- and end places as the number of incoming and outgoing edges the compound activity itself has.

Arc The arc is used to connect a place with an activity or compound activity, to define how the tokens should flow.

A typical example of a diagram in this layer is shown in figure 3.3, where in the first activity, a message *product* should be received, and afterward the product will be accepted or rejected, which will result in paying the money or sending the product back.

Design time As said, the TALL-layer can be used by humans to model the agent's behavior. The other two layers can automatically be created using the behavior model of the TALL-layer by operations called *Petri-fy* and *compile*.

Figure 3.4: Example behavior layer



Run time During run time, the places of this layer will be filled with tokens, according to the current marking of the compiled behavior. In this way, a human can follow the execution of the behavior of the agent in the original TALL layer, while the behavior is actually executed at the compiled behavior layer.

Align time When the agent has automatically changed his behavior, the model at the TALL-layer has to be rebuild, to allow the human to still follow the execution of the behavior of the agent at the TALL-layer.

A better understanding of this mechanism should be achieved after reading the complete chapter.

3.2 Behavior layer

The second layer is the behavior layer. The behavior layer is the layer used by agents for executing and dynamically changing their behavior. The diagrams used in this layer are Behavior Nets as defined in chapter 2.4. A typical example of such a diagram is shown in figure 3.4 on page 25.

Design time The diagram shown in figure 3.4 is actually equivalent to the diagram shown in figure 3.3 on page 24, but the compound activities are "exploded" to activities on one level. Also, all typical TALL-constructs are reduced to Behavior Net constructs. For this reason, the operation of converting a behavior diagram of the TALL-layer to a behavior diagram for the behavior layer is called Petri-fying, as the TALL diagram is reduced to an extension of Petri Nets.

Table 3.1: TALL constructs and their Behavior Net equivalents

TALL model construct	Behavior Net equivalent
Place	Place in P
Start place	Place in P, and i
End place	Place in P, and o
Message place	Place in Pm
Activity	Transition in T
Token road	Arc in Fi or Fo
Message road	Arc in Fi or Fo

Table 3.1 shows how all the TALL model constructs are reduced to get a valid Behavior Net. In TALL, there are no formal definitions of data types, guard expressions, and binding functions. For this reason, we presume that the TALL-model that we want to reduce to a Behavior Net is modeled with the same definitions of data types, guard expressions and binding functions as used in Behavior Nets.

Run time During run time, the places of this layer will be filled with tokens, according to the current marking (or state) of the compiled behavior.

Align time During align time, the agent will adapt her behavior, as will be discussed later on in this report. After dynamically adapting her behavior, the compiled layer explained in the next section has to be recreated by compiling, and the tokens which are currently in the behavior layer will be used to fill the places of the compiled layer with tokens again.

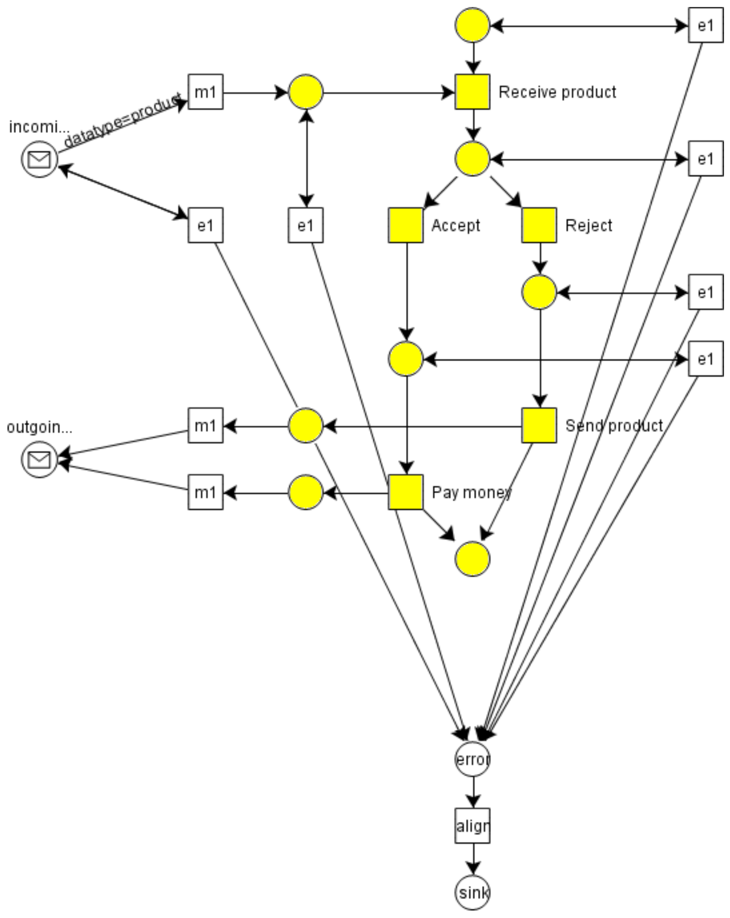
3.3 Compiled behavior layer

The actual layer which will be executed by the engine is the compiled behavior layer. Diagrams used for this layer are Behavior Nets just like with the behavior layer.

Debug and align time When the behavior layer is changed, the compiled behavior layer has to be rebuild, by compiling the behavior layer. Compiling adds extra nodes and edges for message handling and error detection. An example of such a diagram is shown in figure 3.5 on page 27. The colored nodes are the original diagram, as shown in figure 3.4 on page 25. Extra nodes are added to the diagram, with the purpose of collecting data if there is no progress in the execution of the data, and to handle the incoming and outgoing messages. The following extra nodes are added:

- An extra place on which incoming messages will be placed by the engine. This node is called `incomingMsg`.

Figure 3.5: Example compiled layer



- An extra place on which outgoing messages will be placed, and which will be read by the engine, to check for messages to send to other participants in the interaction. This node is called `outgoingMsg`.
- A place called `error`, for collecting error tokens, an transition `align`, in which a dynamic change of the behavior can be initiated, and a place called `sink` in which all error tokens will end up.
- For every place in the behavior, expect the places `error` and `sink`, an extra transition `e` will be added, which will consume the token on that place when the token is not moving for some amount of time (in the example 20). This transition will immediately put back the same token on this place, and sends a token to the `error` place. In this way, the information about the problem can be collected, by putting content in the error tokens telling where the tokens are not moving. The `align` transition can take some appropriate action, based on the content of the tokens it receives.
- For every original incoming message place an extra transition `m` will be added, which connects the new place `incomingMsg` with the original message place. The label of the original message place is added as a constraint to the edge, to ensure the received message will go to the appropriate message place. When an unknown and/or unexpected message type message is received, an error token telling this will be send to the `error` place, by the previously added transition `e`.
- A new start node is added (not shown in the picture), connected with a transition to the original start node, to maintain the property that a Behavior Net has one start node, which has no incoming edges.

Run time During run time, the marking (or state) of this layer will be used to fill the places of the behavior layer and the TALL-layer.

Chapter 4

Modifying Behavior Nets

As most people see the reactivity for external events as the main problem of workflows (and thus for Behavior Nets; received messages can be compared to external events) [14, 15], it is of importance to investigate how Behavior Nets can be modified, to suit them for reactive adapting for e.g. unexpected received messages. In this chapter will be discussed how Behavior Nets can be modified, as in chapter 6 and 7 these techniques will be applied for automatically adapting Behavior Nets.

4.1 Operations preserving global soundness

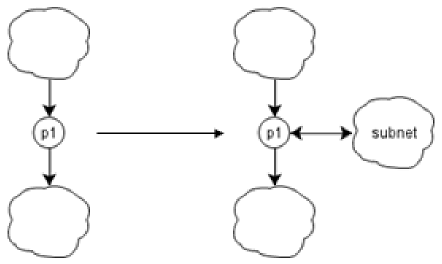
In [6], Van der Aalst describes inheritance preserving transformation operations for private workflows in inter-organizational workflows. When such a operation is applied to a private workflow, this will not disturb the public process. These operations can be applied in the same way for behaviors in interactions as for workflows. Applying the operation will not disturb the interaction, for example by creating deadlocks. This is because the new behavior are subclasses of the old behavior (by projection inheritance, see [6]), and thus will this transformation have no effect on the interaction. The three inheritance preserving transformation operations according to Van der Aalst can be seen in figure 4.1 on the following page.

Transformation operation PP Adding a subnet to a place, which temporarily removes the token from the original place, as seen figure 4.1 (a). This preserves soundness, as long as the new subnet itself is also sound, i.e. when it returns a token to p, there should be no "orphan" tokens left in the new subnet.

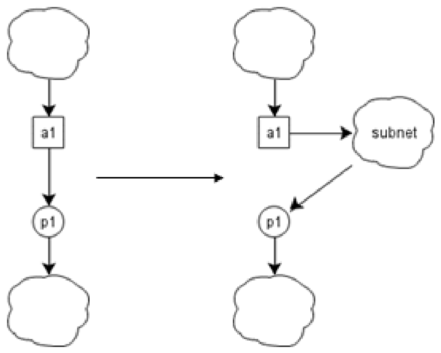
Transformation operation PJ Adding a subnet between a transition and a place (see figure 4.1 (b)). This preserves projection inheritance, because when we abstract away from the added subnet, the behaviors are identical.

Figure 4.1: Transformation operations preserving inheritance

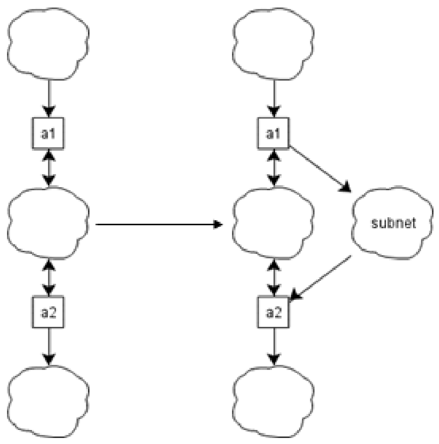
(a) Transformation rule PP



(b) Transformation rule PJ



(c) Transformation rule PJ3



Transformation operation PJ3 Adding a subnet between two transitions (see figure 4.1 (c)). This preserves inheritance for the same reason as with transformation operation PJ.

4.2 Operations preserving local soundness

In [10] Chrzastowski-Wachtel et al. give operations for building dynamic workflows top down. They give five refinement operations (sequential place split, sequential transition split, OR-split, AND-split and loop) which preserve local soundness, and can be used for top-down model development. The inverse of those operations are not given, but it is obvious that they also can be used in bottom-up modeling. Also non-refinement operations are given, for communication and synchronization, which preserve soundness when they are applied under certain conditions. Van der Aalst gives in [5] a set of soundness preserving transformation operations similar to the refinement operations given by Chrzastowski-Wachtel et al.. He gives five pairs of basic transformation operations, each time a operation with its inverse. Below I will briefly discuss the relevant operations for adapting behavior.

Division/aggregation Splitting a transition into two sequential transitions. This is what Chrzastowski-Wachtel et al. call *sequential transition split*. The inverse is called aggregation. This can be seen in figure 4.2 (a).

Specialization/generalization Dividing a transition into two specializations. Chrzastowski-Wachtel et al. call this an *OR-split*. The inverse is called generalization. This can be seen in figure 4.2 (b).

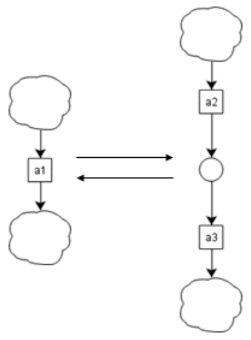
Parallelization/sequentialization Two sequential transitions will be executed in parallel. This same effect can be reached with the *AND-split* and the *sequential place split* of Chrzastowski-Wachtel et al. The inverse is called sequentialization. This can be seen in figure 4.2 (c).

Iteration A transition is replaced by an iteration over another transition. This is similar to the *loop* of Chrzastowski-Wachtel et al. This can be seen in figure 4.2 (d). Also the inverse of the iteration can be applied, albeit it does not have a name.

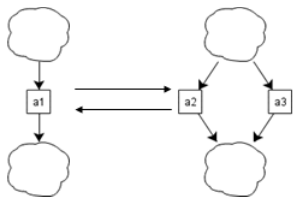
Communication A message place is added to or deleted from a transition, for receiving and sending messages. These operations are a addition to the one mentioned by Van der Aalst and Chrzastowski-Wachtel, because Behavior Nets are used for modeling interactions from an agent perspective. These operations can be seen figure 4.2 (e) and (f).

The proof of the soundness preserving properties of these operations can be found in [5, 10]

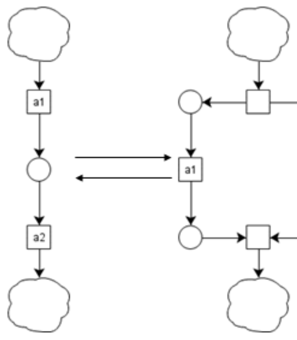
Figure 4.2: Transformation operations preserving soundness



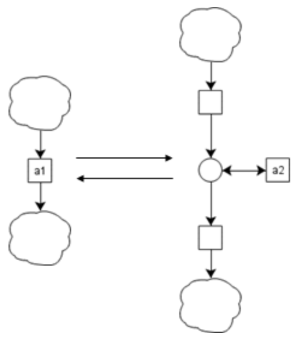
(a) Division/aggregation



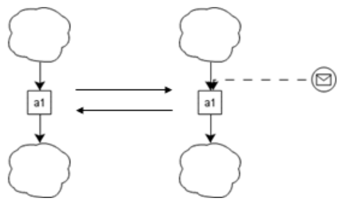
(b) Specialization/ generalization



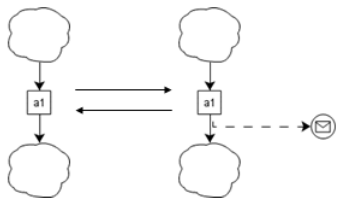
(c) Parallelization/sequentialization



(d) Iteration



(e) Receive message



(f) Send message

Table 4.1: Primitive operations

Operation	Effect
`CreatePlace`(p)	A place p is added to P
`DeletePlace`(p)	A place p is removed from P
`CreateMessagePlace`(p)	A message place p is added to Pm
`DeleteMessagePlace`(p)	A message place p is removed from Pm
`CreateTransition`(t)	A transition t is added to T
`DeleteTransition`(t)	A transition t is removed from T
`CreateArc`($x \in (P \cup Pm)$,$y \in T$)	Add an arc to Fi connecting place x with transition y
`CreateArc`($x \in T$,$y \in (P \cup Pm)$)	Add an arc to Fo connecting transition x with place y
`DeleteArc`($x \in (P \cup Pm)$,$y \in T$)	Removes an arc from Fi connecting place x with transition y
`DeleteArc`($x \in T$,$y \in (P \cup Pm)$)	Removes an arc from Fo connecting transition x with place y
`SetLabel`($x \in (P \cup Pm \cup T)$,l)	Sets the label of x to l
`SetDataType`($x \in Pm$,$y \in \Sigma$)	Sets the data type x may contain to y
`CreateGuard`($x \in Fi$,g)	Adds guard g to arc x
`DeleteGuard`($x \in Fi$,g)	Deletes guard g from arc x
`CreateBinding`(t,b)	Adds binding b to list of bindings of t
`DeleteBinding`(t,b)	Removes binding b from list of bindings of t
`CreateToken`($p \in P$,t)	Adds token t to place p
`DeleteToken`($p \in P$,t)	Deletes token t from place p

4.3 Primitive operations

Based on [16], in Behavior Nets, there are some primitive operations for modifying the net structure. A complete list of these operations is shown in table 4.1. Several of these primitive operations can be combined in order to create a more advanced operation. Applying primitive operation has no guarantee at all to preserve local or global soundness. However, combining them in a way defined in the next section will ensure local soundness.

4.4 Advanced operations

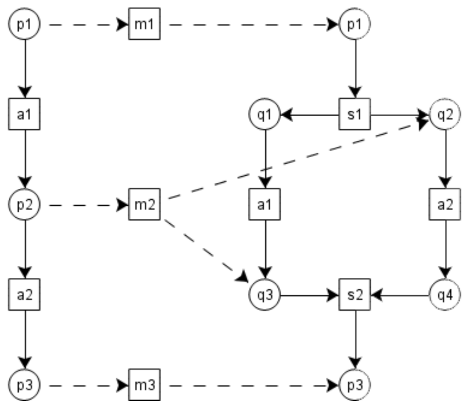
Some of the operations mentioned under operations preserving local soundness are equal to the operations preserving global soundness. However using operations which preserve the global soundness (of which the agent changing its behavior does not know if it exists) are the most safe rules to use. More specifically, if after using a set of rules for adapting the behavior, the new behavior still is a subclass of the old behavior, this has the least chance to mess up the interaction. In addition to the primitive operations, as mentioned above, the advanced operations of table 4.2 on page 35 can also be used to modify the net structure of a Behavior Net, but have the advantage that local soundness is preserved. Note that all these operations can be made by combining a set of primitive operations.

For some of the operations, marked with *, is it not always clear how they can be applied on-the-fly, because of the dynamic change problem [6, 4]. For example, the sequentialization, as mentioned above, cannot be applied for every token-configuration, as it is not always clear where on which place the tokens from the old behavior should be placed. For modeling the change schemes the approach of Ellis et al. [13] is used. By modeling an workflow change as a workflow, it can be exactly defined how to migrate the tokens from the old behavior to the new behavior. This can also be described using the primitive operations shown in table 4.1 on the previous page. For the `receiveMessage`, `notReceiveMessage`, `sendMessage` and `notSendMessage`, nothing needs to be done for the migration, as there is no change in the places, except for the message place, which initially will not have a token. In figure 4.3 on page 36 (a) can be seen how the migration for the operation parallelization can be modeled. Figure 4.3 (b) shows how the same can be formalized using primitive operations.

Table 4.2: Advanced operations

Operation	Effect
`Division`(t,t'_1,t'_2)	Divides a transition t into two sequential transitions t'_1 and t'_2
`Aggregation`(t_1,t_2,t')*	Aggregates two sequential transitions t_1 and t_2 into one transition t'
`Specialization`(t,t'_1,t'_2)	Specializes transition t into two specializations t'_1 and t'_2, chosen by an OR-split
`Generalization`(t_1,t_2,t')	Generalizes two transitions t_1 and t_2 into one generalization t'
`Parallelization`(t_1,t_2)	Puts two sequential transitions t_1 and t_2 into parallel
`Sequentialization`(t_1,t_2)*	Puts two parallel transitions t_1 and t_2 into sequential
`Iteration`(t,t')	Replaces transition t by an iteration over transition t'
`NoIteration`(t,t')	Replaces an iteration over transition t by a transition t'
`ReceiveMessage`(t,$m \in \Sigma$)	Add an incoming message place with an arc connected to t, which receives a message of type m
`NotReceiveMessage`(t,$m \in \Sigma$)	Deletes an incoming message place connected to t, which receives a message of type m
`SendMessage`(t,$m \in \Sigma$)	Add an outgoing message place with an arc connected to t, which sends a message of type m
`NotSendMessage`(t,$m \in \Sigma$)	Deletes an outgoing message place connected to t, which sends a message of type m

Figure 4.3: Migration of old to new behavior



CreatePlace(q1)
CreatePlace(q2)
CreatePlace(q3)
CreatePlace(q4)
CreateTransition(s1)
CreateTransition(s2)
CreateArc(p1,s1)
CreateArc(s1,q1)
CreateArc(s1,q2)
CreateArc(q1,a1)
CreateArc(q2,a2)
CreateArc(a1,q3)
CreateArc(a2,q4)
CreateArc(q3,s2)
CreateArc(q4,s2)
CreateArc(s2,p3)
AddToken(q2,getTokens(p2))
AddToken(q3,getTokens(p2))
DeleteToken(p2,getTokens(p2))
DeleteArc(p1,a1)
DeleteArc(a1,p2)
DeleteArc(p2,a2)
DeleteArc(a2,p3)
DeletePlace(p2)

(a) model of migration	(b) migration by primitive operations

Chapter 5

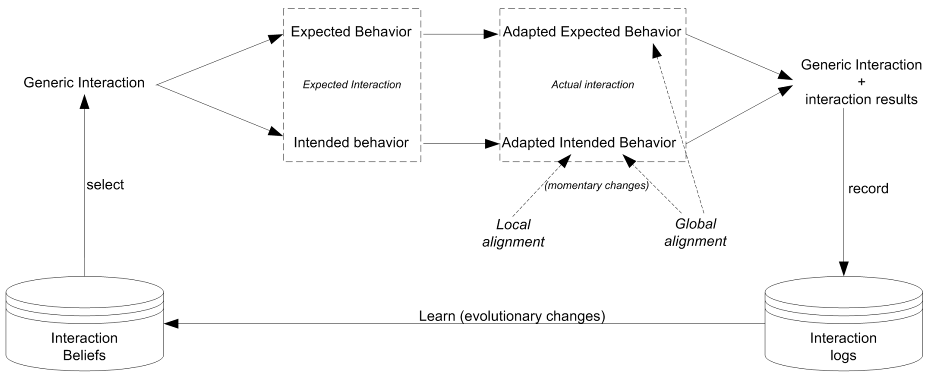
Interactions

When two agents are performing an interaction, they both choose a behavior to execute, which is based on a certain expectation of what the other agent is going to do. Both agents' behaviors will be executed distributively, and the environment will handle the sending and receiving of messages. The agent behavior and its belief of the behavior of the others together form the Interaction Belief. This chapter will discuss the Interaction Belief cycle, and will give some examples of not aligned behaviors.

5.1 The Interaction Belief cycle

The Interaction Belief cycle defines how an agent manages Interaction Beliefs, which consist of the agent's intended behavior, combined which the expected behavior of the other agent interacting with. The cycle consists of *selecting* a behavior for an interaction, *recording* interaction logs, and *learning* new behaviors. Figure 5.1 illustrates the Interaction Belief (IB) cycle. The cycle is

Figure 5.1: The behavior cycle



initiated when the agent will participate in an interaction. The complete cycle consists of the following steps:

1. The agent selects a *generic interaction* from her belief base of Interaction Beliefs, which she finds the most appropriate for the interaction to be performed.
2. This generic interaction defines the *intended behavior* the agent herself is going to execute, and the *expected behavior* of the agent with which the agent is going to interact.
3. After the interaction is executed, the interaction as it was executed including the interaction results will be recorded in the *interaction logs.*
4. New Interaction Beliefs can be learned from the database of interaction logs.

Two different types of change of the intended behavior and interaction can occur during this cycle, *momentary* changes and *evolutionary* changes [4].

Momentary changes Momentary changes will only affect the current interaction, and not the generic known interaction models of the agent. Changes to the intended behavior can be made by the agent herself (local alignment) or by an agent superior to both agents participating in the interaction, e.g. the Deus Ex Machina (global alignment). These two types of alignment will be discussed in the next chapters.

Evolutionary changes Evolutionary changes are changes to the generic Interaction Beliefs, which apply to all future interactions of the agent. From the interaction logs of all interactions executed by the agent can new interaction models be derived. A technique which can be used for this is process mining [7], which extracts process models from event logs, such as the interaction logs. Evolutionary changes of behaviors and generic interactions are outside the scope of this report.

5.2 Examples of Interactions

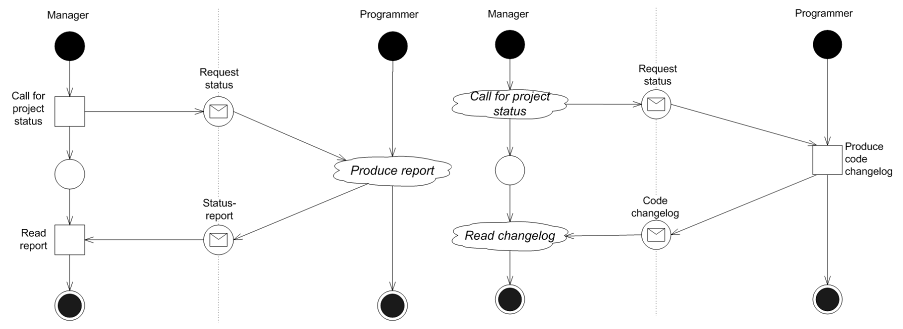
In this section, several examples of interactions in which the intended behaviors of the two agents interaction that are not matching are presented.

5.2.1 Manager and programmer

A typical example is where the manager asks a programmer for a status report about a certain project, but the programmer interprets this by giving a too technical document. This example is illustrated in figure 5.2 on page 39, where the manager expects a short status report, and the programmer gives the change log of the code.

Figure 5.2: Interactions Beliefs of the manager and programmer

(a) Interaction Belief of the manager (b) Interaction Belief of the programmer



This problem can be solved in several ways. If the manager is able to understand the change log, he might be able to subtract the information he needs from the change log. Another possibility would be to ask a third person to subtract this information from the change log. A third option is to try to change the behavior of the programmer, to enforce him to give the short status report the manager is expecting. This last solution is only possible if the manager has the authority to impose that.

In general, one can say if an agent receives another message than expected, she has (at least) these three options:

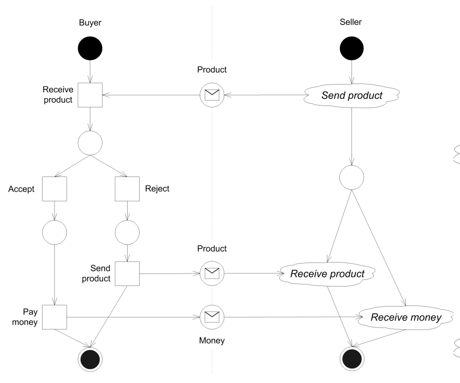
- convert the message yourself
- let someone else convert it for you
- let the agent interacting with send you another message

5.2.2 Buyer and seller

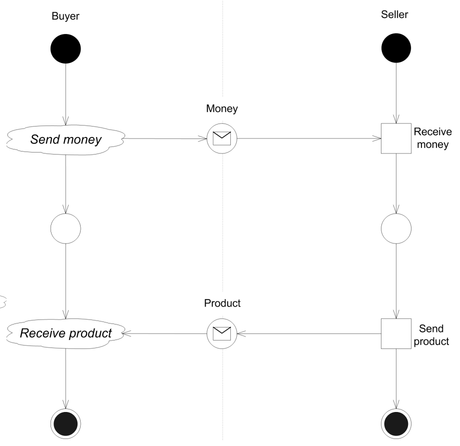
The second example is a more complicated one. A buyer and a seller already agreed on the product the buyer wants to have. But they have different beliefs of how the actual transaction will happen. The example is shown in figure 5.3 on the following page, in which the two Interaction Beliefs are shown. The buyer first expects the product, and if he accepts it, he will send the money, otherwise he wants to send the product back. The seller however, first expects the money, and wants to send the product afterward.

Figure 5.3: Interaction Beliefs of the buyer and seller

(a) Interaction Belief of the buyer



(b) Interaction Belief of the seller



This interaction immediately deadlocks. This can be solved, if one of the two participants is willing to change the order. If they do not want to change order, a third party (Deus ex machina) is needed to solve the conflict. Furthermore, the sending back of the product by the buyer is not expected by the seller, so either the seller should accept a returned product, or the buyer should not send the product back.

Chapter 6

Local alignment

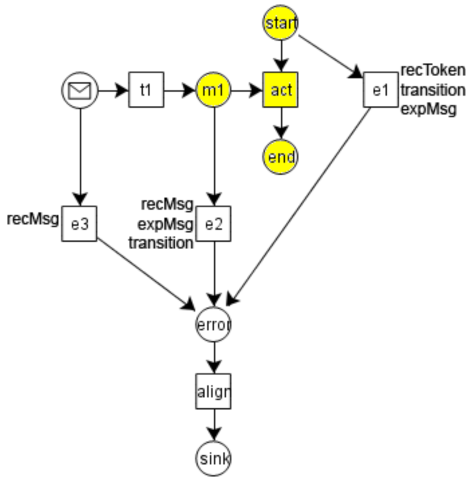
Before two agents start an interaction, they will both individually choose a behavior they are going to execute, based on what they are expecting of the interaction. An interaction however will not successfully terminate, if the behaviors of the agents interacting are not matching. To overcome this problem, agents have to be able to change their behavior on-the-fly, i.e. during the interaction. Alignment policies can be used by agents to change their behavior on-the-fly.

6.1 Collecting problem information

As discussed in section 3.3, the modeled behavior itself is not executed, but the compiled version of it. By compiling, extra nodes are added to the original behavior, for collecting information when there is no progress in the execution of the behavior. This information can be used for selecting a proper alignment policy, which is the best for coping with the problem. Figure 6.1 shows what information can be collected by the extra nodes added when the behavior is compiled. The colored nodes are the nodes of the original behavior, just as in figure 3.5 on page 27. The `align` transition is the transition where all collected information is gathered, and (if necessary) an alignment policy is chosen and executed. The original behavior has only one transition `act` which expects one incoming message `m1`. This transition `act` failed to execute, so all the gathered information is about transition `act` , to make the `align` transition able to choose an alignment policy which will successfully align the behavior around the transition `act`. Despite this simple example, this example shows all ways of how information used for choosing an alignment policy for aligning around a certain transition is collected:

- When a token from a place is not moving (place `start` in this example). In this example, transition `e1` will send information about this to the align transition. It will send the following information:

Figure 6.1: Information collecting



  - `recToken`, short for received token, to inform the align transition that there is a token on that place in the behavior
  - `transition`, the name of the transition for which the `recToken` is required to enable it (`act` in the example)
  - `expMsg`, the data type of the message expected by the transition (in this example the data type of messages `m1` may contain)

- When a known received message is not moving (place `m1` in this example). In this example, transition `e2` will send information about this to the align transition. It will send the following information:

  - `recMsg`, the data type of the received message (in this example the data type which `m1` may contain, which of course equals the data type of the message, because otherwise the message would not be on this place)
  - `expMsg`, the data type of the expected message (also the data type `m1` may contain)
  - `transition`, the name of the transition which is connected to the message place (`act` in the example)

- When an unknown message is received (the message place in this example). In this example, transition `e3` will send information about this to the align transition. It will send the following information:

  - `recMsg`, the data type of the received message

This information is collected separately for every transition, where problems arise.

Table 6.1: Examples of alignment policies

Policy #	Operations
1	`parallelization`(*transition*,*seqTransition*)
2	`notReceiveMessage`(*transition*,*expMsg*) `specialization`(*transition*,Receive *expMsg*, Receive *recMsg*) `receiveMessage`(Receive *expMsg*,*expMsg*) `receiveMessage`(Receive *recMsg*,*recMsg*)

Table 6.2: Variables for defining parameters of operations

Variable	Will be replaced with
transition	the name of the transition
seqTransition	the name of the transition which is sequential placed after *transition*
parTransition	the name of the transition which is parallel with *transitions*
specTransition	the name of the transition which is mutual exclusive to *transition*, and originates from the same specialization
expMsg	the data type of the expected message
recMsg	the data type of the received message

6.2 Alignment policies

An alignment policy is a set of primitive or advanced operations. In our approach, an agent has a set of policies in her knowledge-base from which she can choose when an interaction for example has deadlocked, i.e. when there is no progression anymore in the execution of the behavior. Table 6.1 shows some example policies. As can be seen in the table, the two policies are no more then a ordered set of operations. Policy 1 puts two transitions in parallel. Policy 2 makes a specialization of a transition, which expects a certain message, into two transitions, one that expects the original expected message, and the other that expects the received message. Instead of giving fixed names as parameters for the operations, also variables can be used when defining a policy (or a combination of both). When the policy is actually executed, these variables will automatically be replaced by the appropriate values for that situation. The advantage of this is that in this way a more general policy can be defined, what is not only restricted to a given name of the transition and message types. Table 6.2 shows the list of variables what can be used when defining the parameters, and with what they will be replaced.

6.3 Choosing a policy

With the collected problem information, and other beliefs the agent has, a policy needs to be chosen, in order to overcome the problem with the interaction. For selecting a policy, the agent has a base of training examples, which contain the problem information and the policy which was chosen at that time. A machine learning technique has to be used to generalize over this data, and to make the agent able to choose a policy based on experience. How an agent will choose an alignment policy (or if she will choose one at all) depends on multiple factors. The factors discussed next are: problem information, beliefs about the agent interacting with, and the willingness to change its own behavior.

Problem information Most of the time, a problem will occur, when the agent is not receiving the message she is expecting. It can be that the agent did not receive a message at all, or received a different type of message than expected. If she did receive a message, the type of the received message and other information about the problem can be used as attributes for selecting the proper alignment policy.

Beliefs about the agent interacting with Beliefs about the other agent can be of great importance when choosing an alignment policy. When for example the agent completely trusts the other agent, she might be willing to make more "sacrifices" in changing her behavior than when she distrusts the other agent.

Willingness to change behavior When an agent has very advanced and fine-tuned behaviors, it is not smart to radically change the behaviors because of one exceptional interaction. On the other hand, when the behavior of the agent is still very primitive, changing it a lot could be a good thing to do. So when an agent gets "older", and the behaviors are based on more experience, the willingness to change her behavior will decrease. This approach can be compared with the way humans learn, or with the decreasing of the learning rate over time when training a neural network.

The total process of local alignment is shown in figure 6.2. As the figure shows, the decision algorithm (in our case a neural network) will use the collected problem information and values from the belief base (like trust and willingness) to choose an alignment policy, when there is a problem with executing the Behavior Net. It can however be the case that the agent does not have an appropriate alignment policy for this specific problem. In this case, the agent can trigger escape mode, and ask a human user to give a policy. After the escape mode, the decision algorithm needs to be re-trained, because a new situation in which to apply a certain alignment policy is learned (which is a new training example for the decision algorithm). A human can also teach the agent a new alignment policy, in that case the agent learned a new training example and a new policy. In either way, the agent has learned new ways to overcome her lack

Figure 6.2: Local alignment

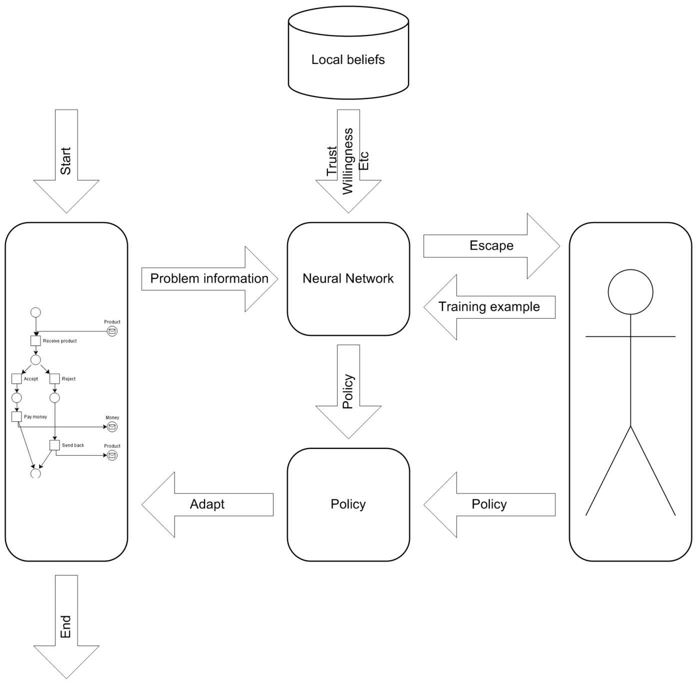


Table 6.3: Example of training data

Example #	1	2	...
transition	Receive money	Receive money	...
seqTransition	Send product		...
parTransition		Send product	...
specTransition			...
recToken	yes	yes	...
recMsg		product	...
expMsg	money	money	...
Trust	1	1	...
Willingness	1	1	...
Policy	1	2	...

of experience. More information about the concept of escape mode can be found in [24]. After the alignment policy is chosen, either by the decision algorithm or by a human, this policy will be executed to adapt the Behavior Net.

The machine learning technique we will use for choosing an alignment policy are neural networks [20]. Neural networks have a practical advantage over decision trees, because not only the selected class (in our case the policy) is returned, but also the activation levels of all classes. In this way, one can know the certainty of the choice of the neural network. In our case, we will compare the activation level of the selected policy with the activation level of the runner-up (the second-best policy). We require that the activation level of the best policy is factor x (with $x > 1$) higher than the runner-up, otherwise, the agent will go into escape mode. In this way, the agent will not execute an alignment policy, when it is questionable that it is suitable for the current problem, and will ask a human for advise. For training the neural network, the open source Java data mining software WEKA is used [28]. An example of the training data with which the neural network is trained is shown in table 6.3. Figure 6.3 shows how the neural network for choosing an alignment policy looks like.

6.4 Example

To demonstrate how these alignment policies could work, this section will give an example, as a proof of concept. In this example, as shown in figure 6.4, the buyer and the seller already agreed on the product the buyer wants to buy, but as seen in the figure, they have different ideas of how the delivery and the payment should go. For the sake of the example, we assume that the behavior of the buyer is very advanced, and thus has no willingness to change her behavior. On the other side, the seller is inexperienced and her behavior is still primitive, so we are looking at the problem how the seller can align her behavior with the buyer, assuming that the seller has trust in the buyer.

When the interaction starts, it immediately deadlocks; the buyer is waiting

Figure 6.3: Neural network

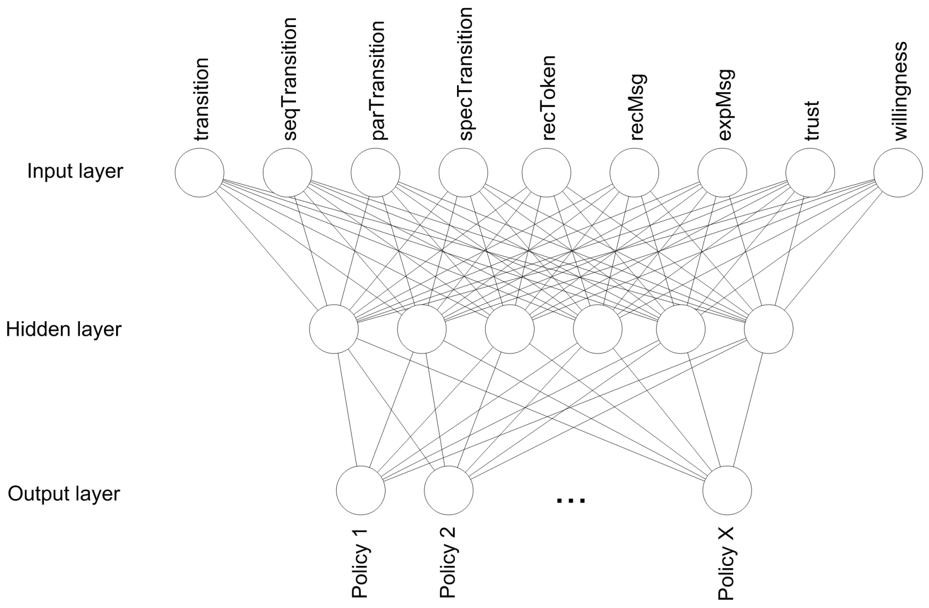
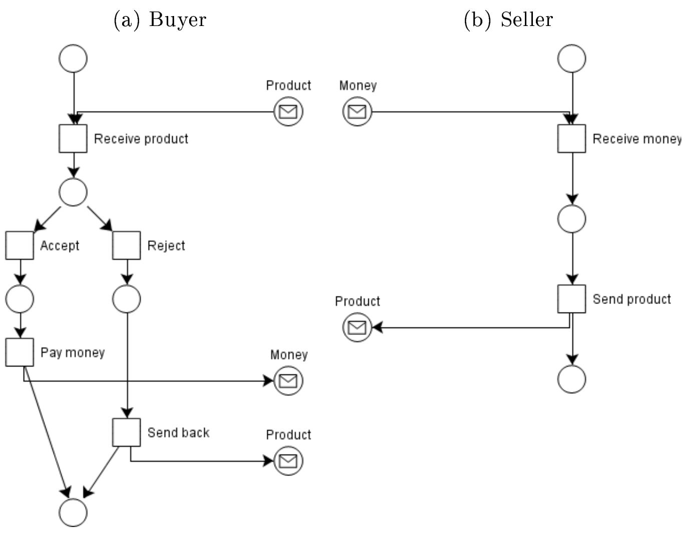


Figure 6.4: Behaviors of buyer and seller



for the product, and the seller is waiting for the money. The seller will collect the information in the way explained in section 6.1. Besides the name of the transition, also `seqTransition`, `parTransition` and `specTransition` (as explained in section 6.2) are looked up. The total collected problem information is shown in table 6.4.

Table 6.4: Collected problem information (1)

Information	Value
transition	Receive money
seqTransition	Send product
parTransition	
specTransition	
recToken	yes
recMsg	
expMsg	Money

This information is used by the neural network to select the appropriate alignment policy (if there is one). For this example, the neural network of figure 6.3 on the previous page is used. This leads to executing policy 1 of table 6.1 on page 43. This means that the seller will place her two transitions in parallel, so she sends the product, and waits for the money in parallel. Hence, by using this alignment policy the behavior of the seller will change to the behavior as seen in figure 6.5 (a) on page 49.

Still the two behaviors are not aligned. If the buyer rejects the product, and sends it back, the seller still does not have the appropriate behavior to handle this; the seller is only expecting the money. When the buyer sends the product back, the seller will collect the problem data as shown in table 6.5 on page 48.

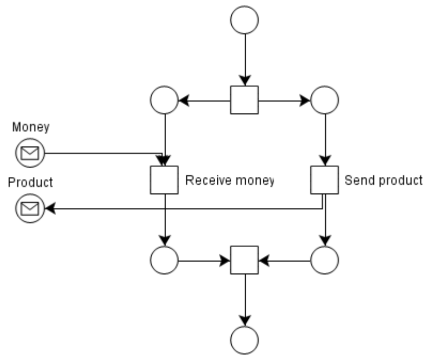
Table 6.5: Collected problem information (2)

Information	Value
transition	Receive money
seqTransition	
parTransition	Send product
specTransition	
recToken	yes
recMsg	Product
expMsg	Money

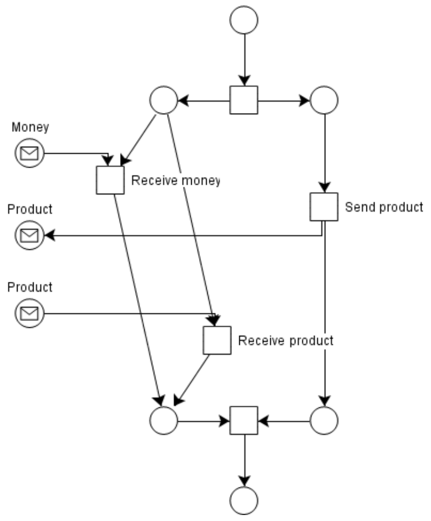
The neural network for selecting an alignment policy is used again, and this time policy 2 of table 6.1 on page 43 is chosen. The seller will divide the transition *receive money* into two mutual exclusive transitions: *receive money* and *receive product*. This will change the behavior as shown in figure 6.5 (a) into the behavior as shown in figure 6.5 (b). The behaviors of the buyer (figure

Figure 6.5: Adapted behavior of seller

(a) First adaptation



(b) Second adaptation



6.4 (a)) and the behavior of the seller (figure 6.5 (b)) are now matching.

Chapter 7

Global alignment

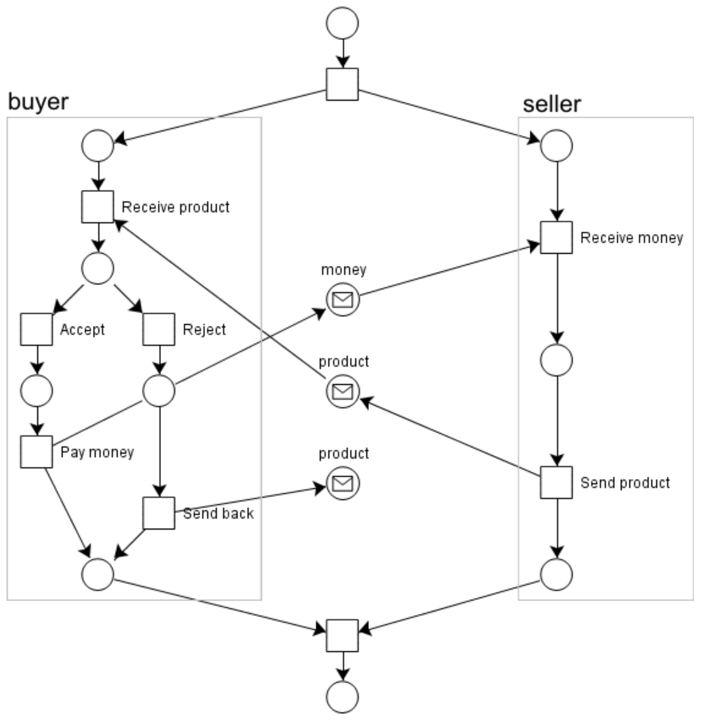
In stead of every agent trying to adapt her own behavior, one agent at a higher level could try to align the intended behaviors of both agents. This has to be a higher level agent, because the agent has to have access to the behaviors of both interacting agents. This way of alignment can be done before the interaction starts, or when the two agents do not manage to align their behaviors on their own. This chapter will discuss a method how this could possibly be done, but will not give a full worked out method as for local alignment. Working out this method can be seen as possible future work.

7.1 Global soundness

A way to check if two behaviors are aligned is to check if the "global" behavior is sound. In other words, one has to make one global behavior out of the two local behaviors, and check if this global behavior is sound. This can be done in in the way as described by Van der Aalst for inter-organizational workflows [1]. In order to create one behavior out of the two, the message places of the two behaviors have to be mapped on each other. Furthermore, an extra input place has to be added, which will be connected with a transition to the original input places of the original behaviors' input places. The same has to be done with the output places. An example of such a behavior is shown in figure 7.1. This is the behavior of the buyer of figure 6.4 (a) on page 47 combined with the behavior of the seller of figure 6.4 (b).

By checking if this behavior (which actually is an interaction) is sound, we know that it will proper terminate (i.e. with only one token in the output place), and that it has no dead transitions. As proved by Van der Aalst in [1], the behavior is sound when the marking of the behavior with only a token in the input place is live and bounded (for the definitions of live and bounded Petri Nets, see section 2.1 on page 14). However, for this, we have to add one extra transition to the behavior, which connects the output node with the input node. This is required for checking the live property.

Figure 7.1: Combined behavior of buyer and seller



If we would verify the behavior of figure 7.1 on the preceding page, it is obvious that it is not sound. There is no reachable state which would enable the transition “Receive product” of the buyer, and the same is true for the transition “Receive money” of the seller. Of course is the same true for the rest of the transitions of the buyer and seller, as they require a token which has to be produced by one of the previous mentioned transitions.

7.2 Aligning the behaviors

The agent aligning both the behaviors can run a simulated interaction with the two behaviors, just like what humans can do with the interaction simulator described in chapter 8. When the interaction is stuck, the agent can collect problem information just as with local alignment, but now the agent gets to sets of problem information, one for each behavior. The agent aligning the behaviors has to choose which of the two behaviors it will adapt. This decision can for example be based on how advanced the behaviors are. The rest of the process could be the same as with local alignment. With the problem data of that behavior, an alignment policy can be chosen, and this policy can be applied on the behavior. After applying the policy, the agent can continue the simulation. If the agent does not know how to align the two behaviors, it can go into escape mode, just as with local alignment.

Chapter 8

Software

During this master graduation project, software has been created to test the Behavior Nets and automatic alignment with policies in interactions. For this purpose, three different applications are created, the *behavior editor*, the *interaction simulator*, and the *alignment editor*. As the names suggest, the behavior editor can be used for creating a Behavior net for an agent for a specific interaction, the interaction simulator can be used to simulate an interaction between agents with different behaviors, and the alignment editor is used for learning the agents new alignment policies and when to use them. The relations between the applications, and their used components can be seen in figure 8.1 on the next page. All this will be discussed in more detail in the upcoming sections of this chapter.

8.1 Software components

This section will describe the major classes as shown in figure 8.1 on the following page, including their major properties and functionality.

8.1.1 Behavior Model

The class `BehaviorModel` is in principle not much more then a list of nodes and a list of edges. The list of nodes consists of objects which are subclasses of the abstract class `Node`, and the list of edges consists of objects which are subclasses of the abstract class `Edge`. Both `Node` and `Edge` are a subclass of `ModelEntity`, which only has one property, namely a `ModelEntityID`. The class diagram in figure 8.2 on the next page shows the relations between the classes that are part of a behavior model. The classes are discussed in more detail in the remainder of this section. The class `BehaviorModel` also has the functionality to save the behavior model as XML, and to reload a saved as XML behavior model. All classes which are part of a behavior model have their functions, for saving and reloading that part of the behavior model to XML. This has the advantage that

Figure 8.1: Class diagram of the main components

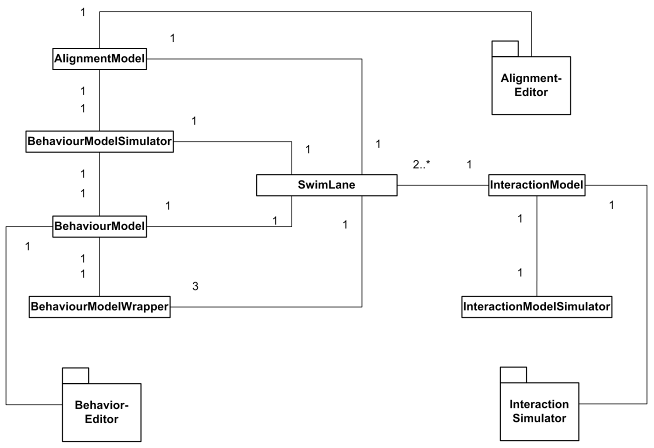
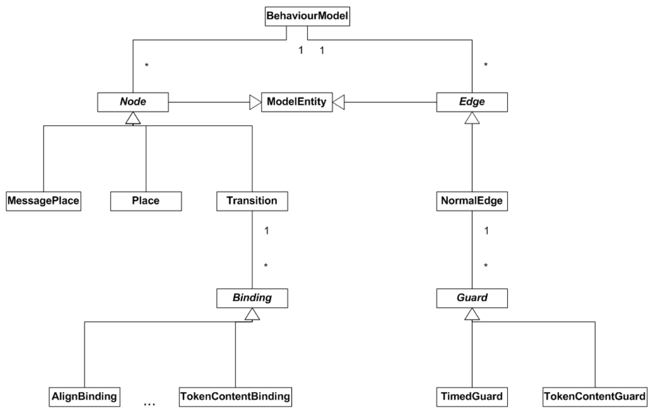


Figure 8.2: Class diagram of the behavior model



new subclasses can easily be added, if you just define in this new class how it can be converted from and to XML. In this way, custom properties can be added to every class.

The nodes

There are a variety of different subclasses of `Node` implemented. They can be divided in nodes which can be part of a Behavior Net, and nodes which are part of the TALL language. For Behavior Nets, the following node classes are implemented:

- `MessagePlace`
- `Place`

- `Transition`

The following node classes for TALL are implemented:

- `Activity`, which is a subclass of `Transition`
- `CompoundActivity`, which contains a `BehaviorModel` itself, and is a subclass of `Transition`
- `DataStore`, which is a subclass of `Place`
- `EndPlate`, which is a subclass of `Place`
- `ExternalProcess`, which is a subclass of `Transition`
- `MessagePlate`, which is a subclass of `MessagePlace`
- `Plate`, which is a subclass of `Place`
- `StartPlate`, which is a subclass of `Place`

All nodes which are a subclass of Transition have a list of bindings, which contains objects which are subclasses of the abstract class `Binding`. These bindings are executed after the transition has consumed the tokens, and before the transition has produced the tokens, as the bindings can alter the content of the produced tokens. The following bindings are implemented at the moment:

- `AlignBinding`, which will start the `AlignmentModel` to modify the behavior on-the-fly
- `ConstantBinding`, which binds a constant value to the tokens which will be produced
- `CopyContentBinding`, which binds all values of the consumed tokens to the tokens which will be produced
- `ShowMessageBinding`, which pop ups a message which can show some of the content of the consumed tokens
- `TokenContentBinding`, which binds one specific value of the consumed tokens to the tokens which will be produced

The edges

There are a variety of different subclasses of `Edge` implemented. They can be divided in edges which can be part of a Behavior Net, and edges which are part of the TALL language. For Behavior Nets, the following edge class is implemented:

- `NormalEdge`, which is just an edge to connect two nodes

The following edge classes for TALL are implemented:

- `DataRoad`, which should be used for connecting a transition with a data store
- `MessageRoad`, which should be used for connecting a transition with a message plate
- `TokenRoad`, which should be used for connecting transitions with the other places

All edges have a list of guards, which contains objects which are subclasses of the abstract class `Guard`. Guards are evaluated before a transition consumes a token from a place, connected with the edge the guard is attached to. In order to let the token be consumed by the transition, the guards must evaluate to true. The following guards are implemented at the moment:

- `TimedGuard`, which lets the transition wait for a certain time, before it can consume the token
- `TokenContentGuard`, which gives a constraint to the content of the token, before it allows the transition to consume it

8.1.2 Swim lane

A `SwimLane` object is no more then the set of objects needed to run a behavior and to visualize it. A swim lane has three `BehaviorModelWrappers`, as it can visualize the behavior on all the three layers, as discussed in section 3. Furthermore, it contains the simulator to control the token flow through the behavior. Finally it contains the alignment model needed for automatic on-the-fly alignment of the behavior model.

8.1.3 Interaction Model

An interaction model is implemented by the class `InteractionModel`. An interaction model is no more then a set of swim lanes, containing all the objects needed for running and visualizing a behavior.

8.1.4 Visualization

The visualization of the behavior model is taken care of by `BehaviorModel-Wrapper`. This class uses the class `BehaviorModel` for getting information about the nodes and edges. A `BehaviorModelWrapper` has functions for auto-arranging the nodes of the behavior model, and for painting them on a panel. For storing the location where to visualize nodes and edges, the `BehaviorModel-Wrapper` has a list of `BehaviorNodeWrappers` and `BehaviorEdgeWrappers`.

8.1.5 Simulation

The class `BehaviorModelSimulator` will manage the 3 layers of the behavior model as discussed in chapter 3. This class will change the marking (the state) of the behavior models on all the three layers, by running the behavior on the compiled behavior layer (i.e. managing the consuming and producing of tokens).

The execution of an interaction is managed by the class `InteractionModel-Simulator`, which uses a set of `BehaviorModelSimulators` to run the behaviors (which it can access through the interaction model, which is the set of swim lanes), and takes care of the exchanging of messages between the behaviors.

8.1.6 Alignment model

The class `AlignmentModel` is connected to a database, which contains the known alignment policies of the agent. Furthermore does the database has a set of training examples, which tell when to apply a certain policy. The class `AlignmentModel` will train a neural network with these training examples, and use this network to choose and execute a specific policy, when the `AlignBinding` has informed the alignment model with the needed problem information.

8.2 The process of alignment

Figure 8.3 on the following page shows how the complete process of alignment works. Not all the function calls are real functions in the software, because from some details is abstracted away, to keep the figure understandable.

The figure is divided in 3 parts, *design time*, *run time* and *align time*. Each part will be discussed in more detail below.

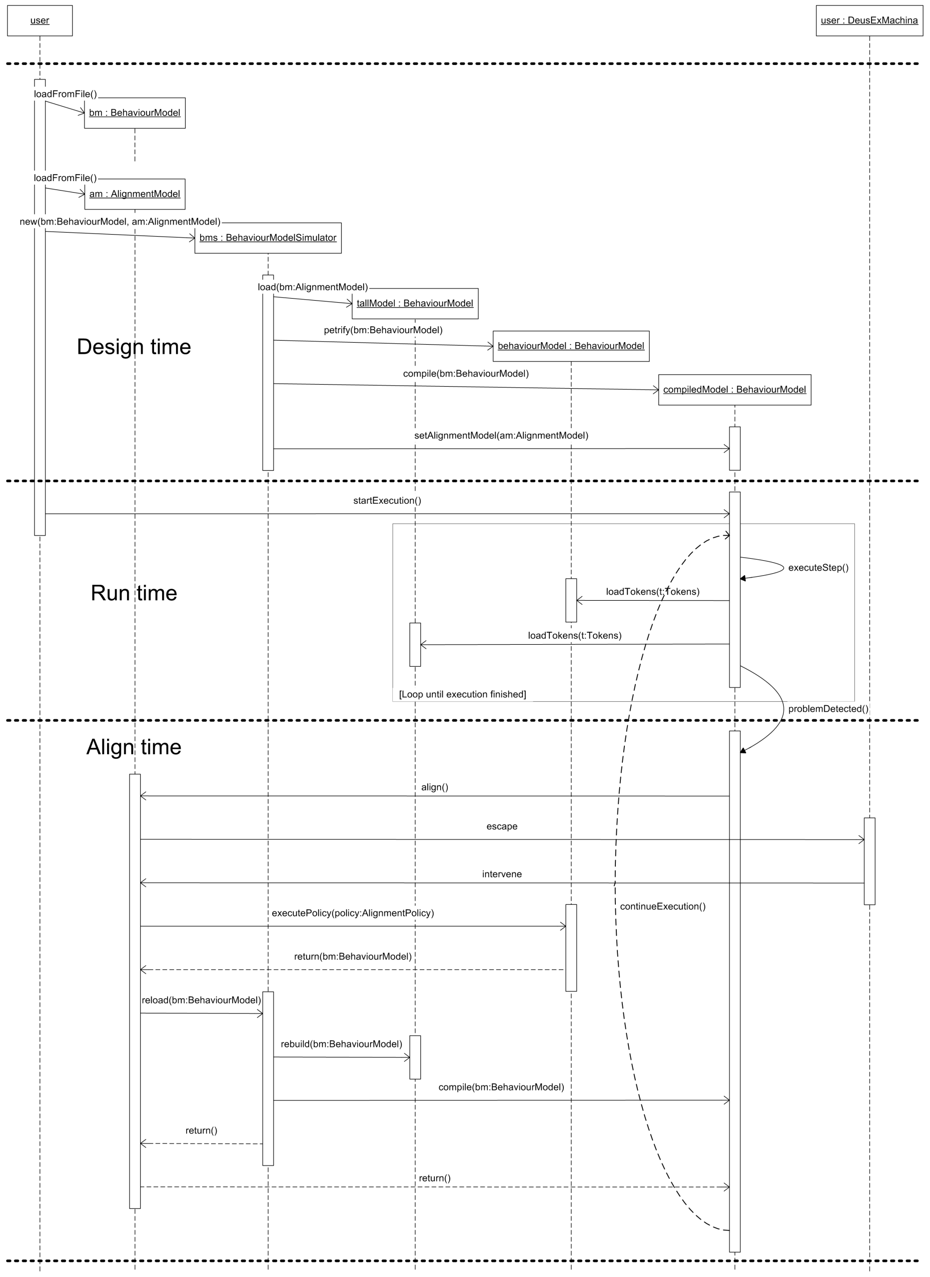
8.2.1 Design time

The sequence diagram of figure 8.3 does not show part of the process of creating the agent behaviors, but only the last part of the design phase, where the behavior and alignment model are loaded, to start running them. A `BehaviorModelSimulator` is created, with the behavior model and alignment model as parameters. This behavior model simulator will create the three layers of behaviors, as discussed in chapter 3 on page 22. The compiled layer has an `align` transition, which needs an alignment model. This is shown in the sequence diagram by the function `setAlignmentModel(...)`.

8.2.2 Run time

When the behaviors of both agents are loaded, the interaction can start. In the sequence diagram, this is visualized by a user which operates the interaction simulator, and calls `startExecution()`. As described in chapter 3, the behavior is executed on the compiled layer level. After each execution step, the behaviors on the other levels are also updated, the TALL layer for visualization purpose,

Figure 8.3: Sequence diagram of the process of alignment


user
user : DeusExMachina
loadFromFile()
bm : BehaviourModel
loadFromFile()
am : AlignmentModel
new(bm:BehaviourModel, am:AlignmentModel)
bms : BehaviourModelSimulator
load(bm:AlignmentModel)
tallModel : BehaviourModel
petrify(bm:BehaviourModel)
behaviourModel : BehaviourModel
Design time
compile(bm:BehaviourModel)
compiledModel : BehaviourModel
setAlignmentModel(am:AlignmentModel)
startExecution()
executeStep()
Run time
loadTokens(t:Tokens)
loadTokens(t:Tokens)
[Loop until execution finished]
problemDetected()
Align time
align()
escape
intervene
continueExecution()
executePolicy(policy:AlignmentPolicy)
return(bm:BehaviourModel)
reload(bm:BehaviourModel)
rebuild(bm:BehaviourModel)
compile(bm:BehaviourModel)
return()
return()

and the behavior layer for the agent (this is the layer on which alignment takes place). When there is a problem detected in the execution of the behavior, the execution will be suspended, and the agent will try to align (this is depicted by the function call `problemDetected()`).

8.2.3 Align time

When there is a problem detected, the `align` transition will call the `align()` function of the alignment model. In the sequence diagram of figure 8.3, the case is handled where the alignment model (which is part of the agent) escapes to the Deus Ex machina for advice. The Deus Ex machina will return an alignment policy the agent has to execute. The agent has now learned a new way to handle this problem type. The alignment model will execute this policy on the behavior layer, and afterward it will reload respectively compile the TALL layer and the compiled layer. The execution of the behavior will now continue again.

8.3 Applications

With the use of the components as mentioned above, three main applications have been created, the *behavior editor* and the *interaction simulator*, and the interaction simulator will use the *alignment editor*, when the agent escapes. This application can also be executed stand alone, but this does not make much sense.

8.3.1 Behavior editor

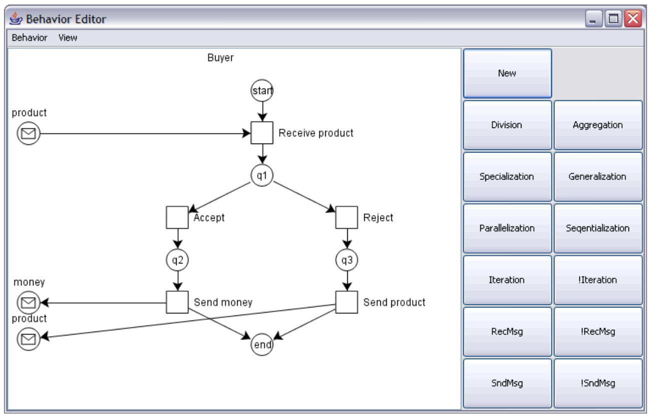
The behavior editor is, as the name suggests, an editor for Behaviors Nets as well as for behaviors made of TALL constructs. The editor is shown in figure 8.4 on the following page. By right-clicking on the behavior with the mouse, a pop-up menu appears for adding and deleting of nodes, edges, bindings and guards. With the buttons on the right, all soundness preserving operations as explained in table 4.2 on page 35 can be used. The right panel also has a "New" button, which does not create an empty Behavior Net, but the smallest Behavior Net possible. This is the Behavior Net with only one transition, and one input and one output place. Starting with this Behavior Net, and only applying the soundness preserving operations of the right panel for creating the complete behavior will always give a sound Behavior Net.

Furthermore, the editor can load and save behaviors from and to XML files.

8.3.2 Interaction simulator

With the interaction simulator, two or more agents who interact can be simulated. For all the simulated agents, a behavior has to be loaded from a XML-file. The class `InteractionModelSimulator` is used for executing the behaviors. A database which will be used by the alignment model can be attached to every

Figure 8.4: The behavior editor



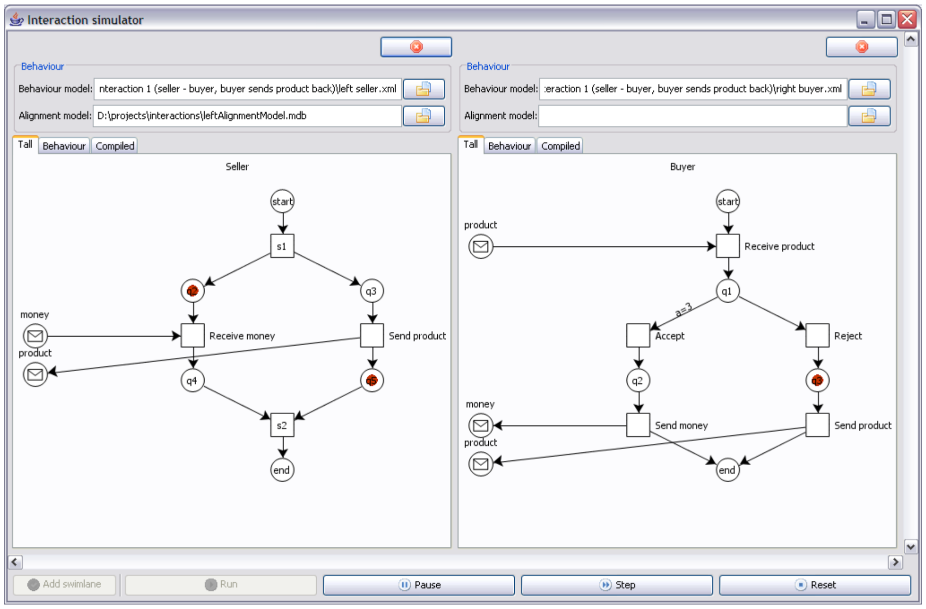
behavior. If a behavior does not have this database, it will not apply any on the fly alignment. With this application can be simulated how an agent would act in AGE in a specific interaction. The interaction simulator is shown in figure 8.5 on page 61.

8.3.3 Alignment editor

In the interaction simulator, a database for the alignment model can be attached to every behavior. When such a database is attached, and a problem is detected in the execution of the behavior, the alignment model will try to adapt the behavior on the fly. If it fails, for example because of the lack of experience, the alignment model will ask for advice of a human, i.e. it will escape. In this case, the human will see the alignment editor, as shown in figure 8.6 on page 61.

In this editor, the human can choose a suitable existing policy, or can build a new policy for this problem. In either case, the agent has gained experience on how to handle a problem like this, and it is less likely that it will escape again in a situation like this.

Figure 8.5: The interaction simulator



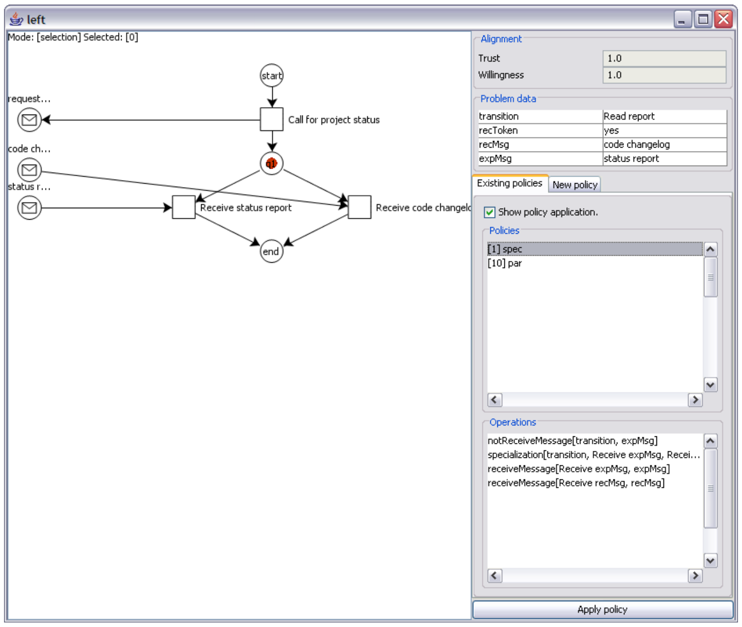
Behaviour model: nteraction 1 (seller - buyer, buyer sends product back)\left seller.xml

Alignment model: D:\projects\interactions\leftAlignmentModel.mdb

Behaviour model: :eraction 1 (seller - buyer, buyer sends product back)\right buyer.xml

Alignment model:

Figure 8.6: The alignment editor



Mode: [selection] Selected: [0]

Trust	1.0
Willingness	1.0

transition	Read report
recToken	yes
recMsg	code changelog
expMsg	status report

☑ Show policy application.

[1] spec
[10] par

notReceiveMessage[transition, expMsg]
specialization[transition, Receive expMsg, Recei...
receiveMessage[Receive expMsg, expMsg]
receiveMessage[Receive recMsg, recMsg]

Chapter 9

Discussion and Future Work

As far as we know, this research is the first attempt to apply this kind of discrete mathematics to anticipatory agents. This approach has the potential to appeal to two research communities: the one oriented towards Business Information Systems development (who apply Petri Net like modeling to BPM and ERP), and also to the growing adaptive agent community.

9.1 Limitations of alignment policies

The technique of alignment policies has several limitations, which will be discussed in this section.

An alignment policy is limited to a certain scope, only the transitions defined by the problem information can be used as variables for the operations of the alignment policy when defining an alignment policy. Of course, other transitions can be used, but these transition names have to be used as constants into the alignment policy. This has the dis-advantage that the policy is not generic anymore, and only suitable for the problem it was designed for. That is why using other transitions than those defined by the problem information is not recommendable. A human should not make an alignment policy for completely redesigning an agent's behavior, but only for smaller adjustments which the agent could later apply themselves in different situations. If a human wants to completely redesign the agent's behavior, (s)he should redesign the behavior directly through intervention, and not with the use of alignment policies.

A second limitation of the alignment policies is the fact that the problem data (what is also used for selecting the alignment policy) can only contain one of the transitions which are in parallel with the transition where the problem is. It could however be the case that there are three transitions in parallel. Using the variable `parTransition` in defining the alignment policy would give a non-deterministic execution of the policy. Building behaviors and alignment policies which give behaviors with three parallel transitions is not recommended when making a behavior which is suitable for automatic alignment. In this case however it is still possible to define alignment policies using constants, but this again will make the alignment policy not generic applicable anymore. The same problem arises, when the problem transition has multiple mutual exclusive transitions of the same generalization (the `specTransition` variable) or when the transition has multiple incoming places (in this case the `recToken` variable does not say so much, as you still do not know if all incoming places do contain a token), or when the transition has multiple incoming message places.

9.2 Possible future work

The research as described in this report raises a number of open questions which could be investigated in future research.

First, the technique of using alignment policies to automatically adapt behaviors should be tested a lot more in practice, and if needed be improved, also to overcome the limitations. Furthermore, we have used a neural network to choose an alignment policy when a problem is detected. It could however be that other machine learning techniques are more suitable for this problem. Furthermore is the neural network not tested with a large set of training examples and alignment policies. At this point we do not know if the configuration of the neural network is optimal, or if it is for example over-trained. This all should be tested in the AGE framework applied in a real life situation.

In this thesis, the concept of global alignment has not been worked out to the level of detail as local alignment. A technique for global alignment has been proposed, based on the used technique for local alignment. It could however be that for global alignment more appropriate techniques exist. Furthermore is there no software implementation for testing global alignment techniques.

Other ways for alignment can also be investigated, like a priori alignment, which can be realized by a superior agent, or even by the agents themselves through a special kind of "pre-alignment interaction" that would entail negotiation.

The described way of automated alignment can align two behaviors of two agents, so that the interaction will complete successfully. This however does not guarantee that the resulting interaction will be efficient. A next step could be a mechanism in which agents can automatically improve an interaction, i.e. make it more efficient. When an agent has knowledge how a certain process could be performed in a more efficient way, it is not clear how this knowledge can be applied to automatically improve an interaction with other agents, because the other agents have to alter their behavior for this purpose. In other words, how can an agent pursue innovation? Furthermore, how does an agent acquire knowledge on how to improve interactions? Can the agent only learn it from somebody else (for instance the Deus Ex machina), or can an agent invent it itself, for example by simulation?

Chapter 10

Conclusion

As shown, in this thesis it is possible to describe a policy for alignment that can be applied when the interaction beliefs of two or more interacting agents are not matching.We introduced an extension of Petri Nets to capture the intended interaction of an agent in a formal way. Furthermore, a mechanism based on a neural network which chooses an appropriate alignment policy with the collected problem information is proposed. When this mechanism fails to choose an alignment policy, a method based on escape is described to overcome this problem, what will learn the agent new ways of alignment. This research can enable predictive agent model execution (agent-based simulation of organizational models) to be more reliable and necessitate less human intervention in terms of alignment.

Bibliography

[1] W.M.P. van der Aalst, Interorganizational Workflows: An approach based on Message Sequence Charts and Petri Nets, *Systems analysis, modelling, simulation*, vol. 37(3), 335-381, 1999

[2] W.M.P. van der Aalst, Structural Characterizations of Sound Workflow Nets, *Computing Science Reports* 96/23, Eindhoven University of Technology, Eindhoven, 1996

[3] W.M.P. van der Aalst, Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques, *Lecture notes in Computer Science*, vol 1806, 161-183, 2000

[4] W.M.P. van der Aalst and S. Jablonski, Dealing with workflow change: identification of issues and solutions, *Computer systems science and engineering*, vol 5, 267-276, 2000

[5] W.M.P. Van der Aalst, Verification of Workflow Nets, *Lecture Notes in Computer Science*, vol 1248, 407-426, 1997

[6] W.M.P. Van der Aalst, Inheritance of Business Processes: A Journey Visiting Four Notorious Problems, *Lecture Notes in Computer Science*, vol 2472, 383-408, 2003

[7] W.M.P. van der Aalst, Business Alignment: Using Process Mining as a Tool for Delta Analysis and Conformance Testing, *Proc. of the 5th Workshop on Business Process Modeling, Development and Support (BPMDS'04), volume 2 of Caise'04 Workshops*, pages 138-145. Riga Technical University, Latvia, 2004

[8] W.M.P van der Aalst, The Application of Petri Nets to Workflow Management, *The Journal of Circuits, Systems and Computers*, vol. 8(1), 21-66, 1998

[9] Lawrence Cabac, Modeling Agent Interaction Protocols with AUML Diarams and Petr Nets, *Diploma thesis*, University of Hamburg, 2003

[10] P. Chrastowski-Wachtel, B. Benatallah, R. Hamadi, M. O'Dell, and A. Susanto, A Top-Down Petri Net-based Approach for Dynamic Workflow Modelling, *Lecture notes in computer science*, vol 2678, pp. 335-353, 2003

[11] B. Ekdahl, Agents as Anticipatory Systems, World Multiconference on Systemics, *Cybernetics and Informatics (SCI'2000) and the 6th International Conference on Information Systems Analysis and Synthesis (ISAS'2000)*, Orlando, USA, July 23-26, 2000

[12] C. Ellis and G. Rozenberg, Dynamic Change Within Workflow Systems, *Proc. Conference on Organizational Computing Systems (COOCS)*, 10-22, 1995

[13] C.A. Ellis and K. Keddara, A Workflow Change is a Workflow, *Lecture notes in Computer Science*, vol 1806, 201-217, 2000

[14] R. Eshuis and R. Wieringa, Comparing Petri Net and Activity Diagram Varaints for Workflow Modelling - A Quest for Reactive Petri Nets, *Lecture Notes in Computer Science*, vol. 2679, 296-315, 2003

[15] R. Eshuis and R.Wieringa, A Comparison of Petri Net and Activity Diagram Variants, *Proc. 2nd Int. Coll. on Petri Net Technologies for Modelling Communication Based Systems*, Berlin, 2001, 93-104

[16] R. Hamadi and B. Benatallah, Recovery Nets: Towards Self-Adaptive Workflow Systems, *Lecture notes in computer science*, vol 3306, pp. 439-453, 2004

[17] D. Harel and R. Marelly, Specifying and executing behavioral requirements: the play-in/play-out aproach, *Softw Syst Model*, vol 2, 82-107, 2003

[18] K. Jensen: An Introduction to the Theoretical Aspects of Coloured Petri Nets, *Lecture Notes in Computer Science*, vol. 803, 230-272, 1994

[19] M. Klein and C. Dellarocas, A Knowledge-based Approach to Handling Exceptions in Workflow Systems, *Computer supported cooperative work*, vol. 9(3/4), 399-412, 2000

[20] Tom M. Mitchell, Machine Learning, WCB/McGraw-Hill, 1997

[21] C.A. Petri, Kommunikation mit Automaten, *PhD thesis*, Institut fur instrumentelle Mathematik, Bonn, 1962

[22] Alberto B. Raposo, Léo P. Magalhães and Ivan L. M. Ricarte, Petri Nets Based Coordination Mechanisms for Multi-Workflow Environments, *Computer systems science and engineering*, vol. 15, 315-326, 2000

[23] A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen and K. Jensen, CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets, *Lecture notes in computer science*, vol 2679, pp. 450-462, 2003

[24] Gijs B. Roest, Nick B. Szirbik, Intervention and Escape Mode, *Lecture notes in Computer Science,* to appear, 2006

[25] K. Salimifard and M. Wright, Petri Net-based modelling of workflow systems: An overview, *European journal of operational research*, vol 134(3), 664-678, 2001, Salimifard

[26] C. de Snoo, Modelling Planning Processes with TALMOD, *Master thesis*, University of Groningen, 2005. Available at: http://tbk15.fwn.rug.nl/tal/files/ThesisCdS.pdf

[27] M. Stuit, Modelling organisational emergent behaviour using compositional role-based interactions in an agent-oriented language, *Master Thesis*, University of Groningen, 2006. Available at: http://tbk15.fwn.rug.nl/marco

[28] Ian H. Witten and Eibe Frank, Data Mining: Practical machine learning tools and techniques, 2nd Edition, Morgan Kaufmann, San Francisco, 2005.

[29] Website: http://tbk15.fwn.rug.nl/tal

Appendix A

EASSS paper

Paper published in *Proceedings of The 8th European Agent Systems Summer School (EASSS 2006) Student Session*, 2006.

Behaviour Alignment as a Mechanism for Interaction Belief Matching

Gerben G. Meyer

The Agent Laboratory, Faculty of Management and Organization, University of Groningen, Nijenborgh 4, 9747 AG Groningen, The Netherlands, +31 (0)50 3638496
gerbenm@gmx.net

Abstract In this paper, a formalism will be presented to define agents' behaviours (as exhibited in agent to agent interactions), by an extension of Petri Nets, and will be shown how behaviours of different agents can be aligned using specific alignment policies. A mechanism is proposed for automatic choosing of an alignment policy by the agent, in order to make the system more reliable, and to reduce the necessary human intervention. Due to the preliminary nature of this work, future directions of research are pointed out.

1 Introduction

When two or more agents have to perform an interaction that is part of a business process instance (BPI) they can either follow an established protocol, or use their own experience and judgement. In the first case, the agents should learn or be instructed about the protocol beforehand. In the second case, there is no protocol enacted, or if it is, does not cover special circumstances that occur in this particular instance. In this case, the agents have to use their own experience, acquired in previous similar kinds of interaction. The agents will build before the interaction an *intended behaviour* (their own course of action) and also will presume what the other agents are doing (*expected behaviour*). These beliefs will form together a mental state what are called the *interaction belief* of the agent in that situation (before the interaction is started).

The intended behaviour of an agent is modelled as a Petri Net. Two intended behaviours of two agents are not always matching, in this case the interaction will not complete successfully. The process of collaboratively changing behaviour for successful matching is called alignment. The focus of this paper will be how behaviour of agents can be formalized, and proposes an alignment method how an agent can change her behaviour for successful matching.

This paper is organised as follows: Behaviour Nets, with which the behaviours of the agent can be formally modelled, will be discussed in section 2. In section 3 will be explained how agents can change their behaviour on-the-fly to match their behaviour with the agent interacting with. The paper ends with a conclusion and discussion.

2 Behaviour Nets

Petri Nets are a class of modeling tools, which originate from the work of Petri [7]. Petri Nets have a well defined mathematical foundation, but also a well understandable graphical notation [9]. Because of the graphical notation, Petri Nets are powerful design tools, which can be used for communication between the people who are engaged in the design process. On the other hand, because of the mathematical foundation, mathematical models of the behaviour of the system can be set up. The mathematical formalism also allows validation of the Petri Net by various analysis techniques.

The classical Petri Net is a bipartite graph, with two kind of nodes, *places* and *transitions*, and directed connections between these nodes called *arcs*. A connection between two nodes of the same type is not allowed. A transition is *enabled*, if every input place contains at least one token. An enabled transition may fire, which will change the current marking of the Petri Net into a new marking. Firing a transition will *consume* one token from each of its input places, and *produce* one token in each of its output places.

2.1 Behaviour net as a Petri net extension

The Behaviour Nets used in this paper are a Petri Net extension, based on Workflow Nets [1], Self-Adaptive Recovery Nets [5] and Coloured Petri Nets [6]. An example of such a Behaviour Net can be seen figure A.2 (a).

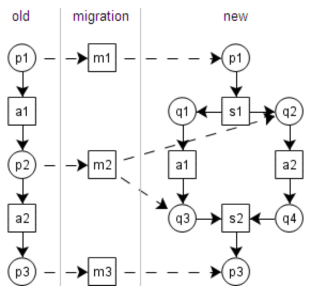
Instead of only places, Behaviour Nets have two types of places, the *normal* places and *message* places. The additional message places are nodes for sending and receiving messages during an interaction. Such a message place is either a place for receiving or for sending messages, it cannot be both. Execution a behaviour is part of an interaction process, the behaviour is created when the interaction starts, and deleted when the interaction is completed. For this reason, the Behaviour Net also has to have one *input* and one *output node*, because the Behaviour Net initially has one token in the input place when the interaction starts, and can be deleted when there is a token in the output place. For each message place is denoted which *data type* it may contain. This is useful for determining on which message place an incoming message has to be placed. Because the two (or more) behaviours in an interaction are distributive executed, messages places of both behaviours cannot be connected directly with each other, as the behaviours do not have to be aligned. Arcs can have a *guard* assigned to them, which expresses what the content of the token has to be, to let the transition consume the token from the place. Transitions can change the content of a token. *Binding functions* defines per transition, what the content of the tokens produced by the transition will be. Bindings are often written as for example: $(T1, < x = p, i = 2 >)$, which means that transition $T1$ will bind value p to x and value 2 to i. The values assigned to the variables of the token can be constants, but can also be values of the incoming token, or values from the knowledge- or belief-base of the agent.

2.2 Operations

In Behaviour Nets, there are some *primitive* operations for modifying the net structure, such as adding and deleting places, transitions, arcs and tokens. Besides the primitive operations, there is a set of more *advanced* operations, which also preserve local soundness. By preserving local soundness is meant that after applying the operation, an execution of the behaviour will still terminate properly, if the behaviour also terminated properly before the operation. The message places Pm are not taken into account when determining local soundness. Local soundness refers to a sound behaviour, to make the distinction with a sound interaction, which will be referred to as global soundness. More information about soundness can be found in [2]. The used set of advanced operations are:

- `division` and `aggregation`*, which divides an transition into two sequential transitions, and vice versa,
- `parallelization` and `sequentialization`*, which puts two sequential transitions in parallel, and vice versa,
- `specialization` and `generalization`, which divides an transition into two mutual exclusive specializations, and vice versa,
- `iteration` and `noIteration`, which replaces an transition with an iteration over a transition, and vice versa,
- `receiveMessage` and `notReceiveMessage`, which adds or deletes an incoming message place,
- `sendMessage` and `notSendMessage`, which adds or deletes an outgoing message place.

Figure A.1: Migration of old to new behaviour



For some of the operations, marked with *, is it not always clear how they can be applied on-the-fly, because of the dynamic change problem [3]. For example, `sequentialization`, as mentioned above, cannot be applied for every token marking, as it is not always clear on which places the tokens from the old behaviour should be placed, when migrating to the new behaviour. For modeling the migrations the approach of Ellis et al. [4] is used. By modeling a behaviour change as a Petri net, it can be exactly defined how to migrate the tokens from the old behaviour to the new behaviour. Note that advanced operations can also be described using the primitive operations. For the `receiveMessage`, `notReceiveMessage`, `sendMessage` and `notSendMessage`, nothing needs to be migrated, as there is no change in the places, except for the message place, which initially don't contain a token. In figure A.1 can be seen how the migration for the operation `parallelization` can be modelled.

3 Aligning Behaviours

Before two agents start an interaction, they will both individually choose a behaviour they are going to execute, based on what they are expecting of the interaction. An interaction however will not terminate, if the behaviours of the agents interacting are not matching. To overcome this problem, agents are able to change their behaviour on-the-fly, i.e. during the interaction. Alignment policies are used by agents to change their behaviour on-the-fly.

3.1 Alignment policies

An alignment policy is a set of primitive or advanced operations. In our approach, an agent has a set of policies in his knowledge-base from which she can choose when an interaction for example has deadlocked, i.e. when there is no progression anymore in the execution of the behaviour. How an agent will choose an alignment policy (or if she will choose one at all) depends on different factors. The factors discussed next are: kind of problem, beliefs about the agent interacting with, and the willingness to change it's own behaviour.

Kind of problem Most of the time, a problem will occur, when the agent is not receiving the message she is expecting. It can be that the agent did not receive a message at all, or received a different type of message than expected. If she did receive a message, the type of the received message and other factors of the kind of the problem can be used as attributes for selecting the proper alignment policy.

Beliefs about the agent interacting with Beliefs about the other agent can be of great importance when choosing an alignment policy. When for example the agent completely trusts the other agent, she might be willing to make more "sacrifices" in changing her behaviour than when she distrusts the other agent.

Willingness to change behaviour When an agent has very advanced and fine-tuned behaviours, it is not smart to radically change the behaviours because of one exceptional interaction. On the other hand, when the behaviour of the agent is still very primitive, changing it a lot could be a good thing to do. So when an agent gets "older", and the behaviours are based on more experience, the willingness to change her behaviour will decrease. This approach can be compared with the way humans learn, or with the decreasing of the learning rate over time when training a neural network.

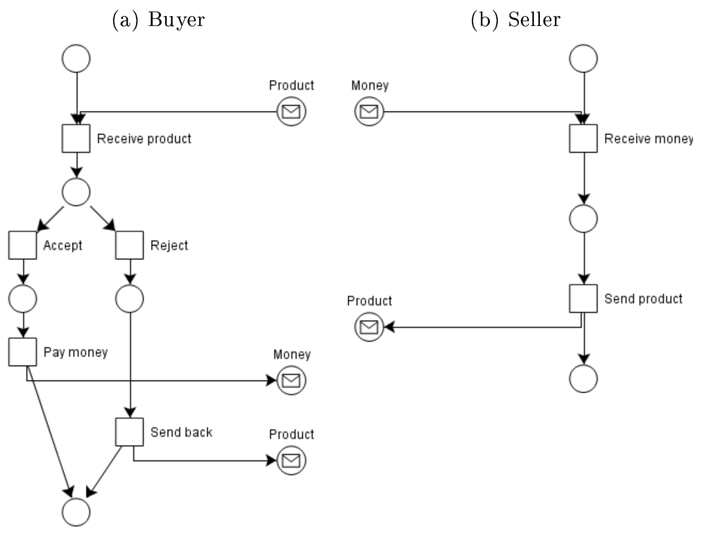
As all this still needs research for what is the best way is to make the decision, a possibility would be to use a decision tree, build on experiences of the use of the alignment policies in previous interactions. A new agent won't have any alignment policies, or experience applying them. When an agent does not know how to handle a certain problem, it can go into escape mode, to learn new ways to overcome her lack of experience. More information about the concept of escape mode can be found in [8].

3.2 Example - Proof of concept

As an example how these alignment policies could work, a small example is given, as a proof of concept. In this example, as shown in figure A.2, the buyer and the seller already agreed on the product the buyer wants to buy, but as seen in the figure, they have different ideas of how the delivery and the payment should go. For the sake of the example will be assume that the behaviour of the buyer is very advanced, and thus has no willingness to change her behaviour. On the other side, the seller's behaviour is still primitive and unexperienced, so we are looking at the problem how the seller can align her behaviour with the buyer, assuming that the seller has trust in the buyer.

When the interaction starts, it immediately deadlocks; the buyer is waiting for the product, and the seller is waiting for the money. A way to overcome this problem would be for the seller to send the product and wait for the money in parallel. So, by using an alignment policy based on the operation `parallelization` the behaviour of the seller changes to the behaviour as seen

Figure A.2: Behaviours of buyer and seller



in figure A.3 (a), and the interaction can continue. However, if the buyer rejects the product, and sends it back, the seller still doesn't have the appropriate behaviour to handle this, because the seller is waiting for the money. In case the seller receives the product back, but when she is expecting the money, the seller could use an alignment policy based on the operation `specialization` to overcome this problem, which divides the receive money transition into two separate transitions, *receive money* and *receive product*. The resulting behaviour can be seen in figure A.3 (b). The behaviours of the buyer (figure A.2 (a)) and the behaviour of the seller (figure A.3 (b)) are now aligned, and thus matching.

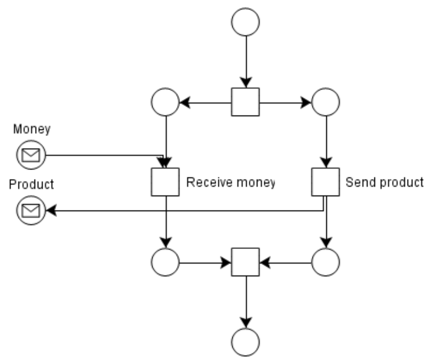
4 Discussion and Conclusion

This research is preliminary. Our research team [10] is developing agents via simulation-games, where the behaviour of the software agents is captured from the experts players from business organisations. These human experts can describe their intended behaviour, in terms of activities and local goals, but also can describe the behaviour they expect from the other agents in the game. Each time an agent cannot find a local solution for a mismatch during an interaction, it can defer control to a higher authority (higher level agent, typically a human). But to minimize the necessary human intervention, the need for better automatic alignment mechanisms becomes very relevant.

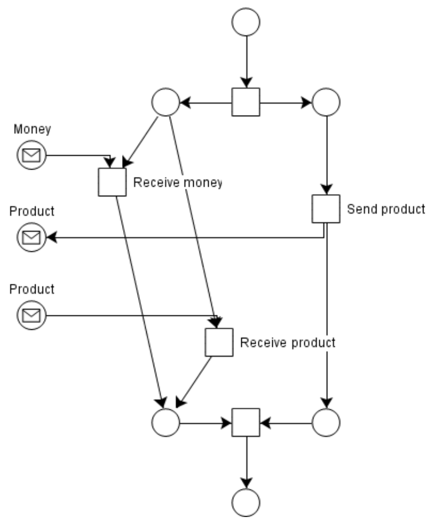
As we have shown, it is possible to describe a policy for alignment that can be applied when the interaction beliefs of two or more interacting agents are not

Figure A.3: Adapted behaviour of seller

(a) First adaptation



(b) Second adaptation



matching. We introduced an extension of Petri Nets to capture the interaction beliefs and also a mechanism to choose the appropriate policy that adapts the beliefs from one agent perspective. This research can enable predictive agent model execution (agent-based simulation of organisational models) to be more reliable and necessitate less human intervention in terms of alignment.

Bibliography

[1] W.M.P. van der Aalst. Verication of WorkowNets. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of Lecture Notes in Computer Science, pages 407-426. Springer-Verlag, Berlin, 199

[2] W.M.P. van der Aalst, Interorganizational Workflows: An approach based on Message Sequence Charts and Petri Nets, *Systems analysis, modelling, simulation*, vol. 37(3), 335-381, 1999

[3] W.M.P. van der Aalst and S. Jablonski, Dealing with workflow change: identification of issues and solutions, *Computer systems science and engineering*, vol 5, 267-276, 2000

[4] C.A. Ellis and K. Keddara, A Workflow Change is a Workflow, *Lecture notes in Computer Science*, vol 1806, 201-217, 2000

[5] R. Hamadi and B. Benatallah, Recovery Nets: Towards Self-Adaptive Workflow Systems, *Lecture notes in computer science*, vol 3306, pp. 439-453, 2004

[6] K. Jensen. An Introduction to the Theoretical Aspects of Coloured Petri Nets. In J.W. de Bakker and W.-P. de Roever, editors, *A Decade of Concurrency, Reflections and Perspectives*, volume 803 of Lecture Notes in Computer Science. Springer-Verlag, 199

[7] C.A. Petri. Kommunikation mit Automaten. PhD thesis, Institut f¨ur instrumentelle Mathematik, Bonn, 1962

[8] Gijs B. Roest, Nick B. Szirbik, Intervention and Escape Mode, *Proc. of AOSE workshop*, AAMAS'06 Conference, Hakodate, Japan, to be published in LNCS, http://tbk15.fwn.rug.nl/tal/downloadle.php?id=3

[9] K. Salimifard and M. Wright, Petri net-based modelling of workflow systems: An overview, *European journal of operational research*, vol 134(3), 664-678, 2001, Salimifard

[10] http://tbk15.fwn.rug.nl

Appendix B

ABiALS paper

Paper published in *Proceedings of The 3rd Workshop on Anticipatory Behavior in Adaptive Learning Systems (ABiALS 2006)*, 2006.

Behaviour Alignment as a Mechanism for Anticipatory Agent Interaction

Gerben G. Meyer[1], Nick B. Szirbik[2]

[1]The Agent Laboratory, Faculty of Management and Organization, University of Groningen, Nijenborgh 4, 9747 AG Groningen, The Netherlands, +31 (0)50 3638496

gerbenm@gmx.net

[2]Dept. Information Systems, Faculty of Management and Organization, University of Groningen, Landleven 5, PB 800, 9700 AV Groningen, +31 (0)50 363 {8125 / 8497}

n.b.szirbik@rug.nl

Abstract In this paper, we present a formalism to define agents' behaviours (as exhibited in agent to agent interactions), by an extension of Petri Nets, and show how behaviours of different agents can be aligned using specific alignment policies. We explain why these agents are anticipatory, and the link between Business Information Systems and anticipatory systems is elaborated. A mechanism is proposed for automatic choosing of an alignment policy by the agent, in order to make the system more reliable, and to reduce the necessary human intervention. Due to the preliminary nature of this work, future directions of research are pointed out.

1 Introduction

In the anticipatory system research community, the agent based computing area is considered a promising one. However, there is little interest yet in applying the anticipatory agent concept in a real setting. Seminal work of Davidsson, Astor and Ekdahl [4], pointed out that agents can be characterised as agents when their acting can be described by a social theory. We argue in this paper that business organisation are in fact anticipatory systems themselves. Especially when these use an information system (usually called BIS - Business Information System). Our research group [14] is investigating novel agent-based architectures and development frameworks [10]. We recognise the importance of the anticipatory system concept in this context and position our models of organisations in the initial definition of Rosen ([11], page 339):

> "We tentatively defined the concept of an anticipatory system: a system containing a predictive model of itself and/or of its environment, which allows it to change state at an instant in accord with the models prediction to a latter instant."

In this paper, which should be seen as a position paper, presenting preliminary research, we investigate how the anticipatory ability of a single agent can be expressed as an interaction belief and also the way this belief can be changed. We will describe a policy for alignment that can be applied when the interaction beliefs of two or more interacting agents are not matching. We will introduce an extension of Petri Nets to capture the interaction beliefs and also a mechanism to choose the appropriate policy that adapts the beliefs from one agent perspective. From the anticipatory systems perspective, this research can enable predictive agent model execution (agent-based simulation of organisational models) to be more reliable and necessitate less human intervention in terms of alignment.

This paper is organised as follows: the rest of the introduction makes the link between BIS and anticipatory systems and gives the motivation for this line of research, section 2 deals with the formalisation of interaction beliefs of agents as Behaviour Nets, section 3 shows how Behaviour Nets of different agents can be aligned by using a specific policy. The paper concludes by discussing the standing issues and questions, pointing towards future research lines.

1.1 Motivation

Business information systems have evolved from a data centric perspective to a process centric perspective. The role of these systems is to support human activity in a business organisation. They support at a basic level information storage and retrieval, information flows and information processing. At a higher level they support human decision making. Depending on the time horizon, the decision can be related to operational management (day-to-day activities), tactical planning (week/month projections), strategic decisions (month/year projections), and even policy implementation (very long term).

The move from data centric to process centric systems did not change the centralistic nature of these systems. The way the system is designed and used ascribes to the notion that there exists an external observer that is able to investigate and understand the processes within the organisation. These processes can be identified in a semantic sense and modelled in a syntactic sense, that is, models of the processes can be described in a (semi) formal language. These models can be used to implement systems that support the actors that execute the process in the organisation.

The actors that are executing the organisations' processes have only local, often conflicting views. If the system is to be designed and implemented by allowing local and different models of the participating actors, a distributed, agent-oriented approach is more suitable. Agent-based modelling and agent-software engineering have been very popular in the last decade and paved new avenues for the development of the business systems of tomorrow. However, as correctly pointed out by Ekdahl [5], the lack of a strict definition of an agent and a clear view about what exactly agent software engineering is, many development processes tend to be in name agent-oriented, in reality, they can be just classified as purely reactive systems. He also states:

> "More sophisticated anticipatory systems are those which also contain its own model, are able to change model and to maintain several models, which implies that such systems are able to make hypotheses and also that they can comprehend what is good and bad."

One can infer from this statement that true agent systems are only those that have a clear anticipatory ability, both at the level of the individual agents themselves, and also at the whole multi-agent system. The ability to reason about a plan in an organisation is usually realised via humans. If one tries to simulate a planning organisation, a typical barrier is the evaluation of the plans. Such simulation tend to become interactive games, where the "players" (i.e. the expert planners) are becoming decision makers that select the "best" plan. Various plan selection mechanisms can be enacted, but these are usually just models of the behaviour of the players. In a monolithic, centralistic ERP system for example, this will be implemented as a single utility function that characterises the whole organisation, which makes explicit the criteria against which a prospective plan is checked. In reality, many expert players are co-operating with the system to adjust and decide for the best plan. The overall behaviour of the organisation (in terms of planning) is just emerging as a combined behaviour of the experts and the system that supports them.

This observation leads to the natural conclusion that it is better to enact decision support structures that mimic the distributed nature of this environment. Attempts to model and implement agent-oriented support for planning and other business processes are still in their infancy, but even simple implementations of crude multi-agent architectures show a higher degree of adaptiveness and flexibility.

1.2 Our approach towards anticipatory agents

Our research team is developing agents via simulation-games [14], where the behaviour of the software agents is captured from the expert players. These human experts can describe their intended behaviour, in terms of activities and local goals, but also can describe the behaviour they expect from the other agents in the game. These behaviours can be simplified and formally described. From a local perspective the *intended behaviour* of self and the *expected behaviour* of others can be seen as a specific *interaction belief* of that agent. The organisations' processes can be viewed as a set of running interactions. Each interaction is executed by the agents that play the roles that define the interaction and the execution depends on the (local) interaction beliefs. If the agents have consistent beliefs, a coherent execution of the interaction will take place. In an environment where human agents are playing the roles, slight (or even severe) misalignment of these behaviours can be solved by the capacity of the humans to adapt to misunderstandings and information mismatch.

Agents (as humans) develop over time a large base of interaction beliefs, which allow them to cope with a wide range of interaction situations. This is why the organisational processes can be carried out in most contexts and exceptional situations. In these, the monolithic and centralistic support of BPM becomes a problem in itself, needing roll-back procedures and "backdoor" interventions. When using an agent-oriented approach, in order to solve the exceptions that occur but have no resolution beliefs implemented in the software agents, a "escape/intervention" [10] mechanism can be used. Each time an agent cannot find a local solution for a mismatch during an interaction, it can defer control to a higher authority (higher level agent, typically a human). Therefore, such a system will never block, supporting the humans up to the levels it has been programmed to do, but leaving the humans to intervene when the situation is too complex for them to solve.

Interaction beliefs are local anticipatory models. These describe future possible states in a specific interaction from a local perspective of an agent. In an organisation, an agent can play various roles by using her "experience" (interaction beliefs that have proved successful in the past), but can also build new ones, depending on the context. Continuous enactment of interaction leads to whole process enactment. In a software multi-agent system, if the captured behaviours are not matching in a given context, the agents will revert to humans. Of course, this can decrease the performance of the system - in terms of support and/or automation - to unacceptable levels. Software agents should be able also to adjust their behaviour in an anticipatory way. There are two ways to tackle behaviour mismatches:

- there is a superior of the two agents that can align and impose a common interaction behaviour that is sound, by having full access to the interaction beliefs of the agents. This can happen before the interaction starts
- each agent is trying to align her behaviour on-the-fly, having only local information

In the next two sections, we describe a method to implement the second choice, by using a representation of the behaviour in terms of Behaviour Nets and a mechanism based on "alignment policies". We considered that the first choice is "less anticipatory", in the sense that only if viewed from a larger perspective (the system is formed by the participating agents, plus the superior agent -

we call this a deus ex machina) becomes a system that investigates a potential scenario for the future. In the "on the fly" mechanism, the anticipatory system is the individual agent who tries to align its behaviour, based on the limited information she has about the interaction execution.

2 Behaviour Nets

Petri Nets are a class of modeling tools, which originate from the work of Petri [9]. Petri Nets have a well defined mathematical foundation, but also a well understandable graphical notation [12]. Because of the graphical notation, Petri Nets are powerful design tools, which can be used for communication between the people who are engaged in the design process. On the other hand, because of the mathematical foundation, mathematical models of the behaviour of the system can be set up. The mathematical formalism also allows *validation* of the Petri Net by various analysis techniques.

The classical Petri Net is a bipartite graph, with two kind of nodes, *places* and *transitions*, and directed connections between these nodes called *arcs*. A connection between two nodes of the same type is not allowed. A transition is *enabled*, if every input place contains at least one token. An enabled transition may fire, which will change the current marking of the Petri Net into a new marking. Firing a transition will *consume* one token from each of its input places, and *produce* one token in each of its output places.

2.1 Definition of Behaviour Nets

In the following, the formal definition of Behaviour Nets is given, which is a Petri Net extension, based on Workflow Nets [1], Self-Adaptive Recovery Nets [7] and Coloured Petri Nets [8]. An example of such a Behaviour Net can be seen figure B.2 (a).

Definition of Behaviour Nets

A Behaviour Net is a tuple $BN = (\Sigma, P, Pm, T, Fi, Fo, i, o, L, D, G, B)$ where:

- Σ is a set of data types, also called colour sets
- P is a finite set of places
- Pm is a finite set of message places (such that $P \cap Pm = \emptyset$)
- T is a finite set of transitions
- $Fi \subseteq ((P \cup Pm) \times T)$ is a finite set of directed incoming arcs
- $Fo \subseteq (T \times (P \cup Pm))$ is a finite set of directed outgoing arcs (such that $Fi \cap Fo = \emptyset$)
- i is the input place of the behaviour with $\bullet i = \emptyset$ and $i \in P$
- o is the output place of the behaviour with $o\bullet = \emptyset$ and $o \in P$

- $L : (P \cup Pm \cup T) \to A$ is the labeling function where A is a set of labels
- $D : Pm \to \Sigma$ denotes which data type the message place may contain
- G is a guard function which is defined from Fi into expressions which must evaluate to a boolean value
- B is a binding function defined from T into a set of bindings b, which binds values (or colours) to the variables of the tokens

The set of types Σ defines the data types tokens can be, and which can be used in guard and binding functions. A data type can be arbitrarily complex, it can be for example a string, an integer, a list of integers, or combinations of variable types.

The places P and Pm and the transitions T are the nodes of the Behaviour Net. All these three sets should be finite. The extension of classical Petri Nets is the addition of the set Pm which are nodes for sending and receiving messages during an interaction. Such a message place is either a place for receiving or for sending messages, it cannot be both.

Fi and Fo are the sets of directed arcs, connecting the nodes with each other. An arc can only be from a place to a transition, or from a transition to a place. By requiring the sets of arcs to be finite, technical problems are avoided, such as the possibility of having a infinite number of arcs between two nodes.

Execution a behaviour is part of an interaction process, the behaviour is created when the interaction starts, and deleted when the interaction is completed. For this reason, the Behaviour Net also has to have one input and one output node, because the Behaviour Net initially has one token in the input place when the interaction starts, and can be deleted when there is a token in the output place.

With function L, a label can be assigned to every node. This has no mathematical of formal purpose, but makes the Behaviour Net better understandable in the graphical representation.

Function D denotes which message place may contain which data type. This is useful for determining which message place an incoming message has to be placed on. Because the two (or more) behaviours in an interaction are distributively executed, message places of both behaviours cannot be connected directly with each other, as the behaviours do not have to be aligned.

Function G is the guard function, which expresses what the content of a token has to be, to let the transition consume the token from the place. Function G is only defined for Fi, because it makes no sense to put constraints on outgoing edges of transitions. In other words, this function defines the preconditions of the transitions.

Transitions can change the content of a token. Binding function B defines per transition, what the content of the tokens produced by the transition will be. Bindings are often written as for example: $(T1, < x = p, i = 2 >)$, which means that transition $T1$ will bind value p to x and value 2 to i. The values assigned to the variables of the token (which data type must be in Σ) can be constants, but can also be values of the incoming token, or values from the knowledge- or belief-base of the agent.

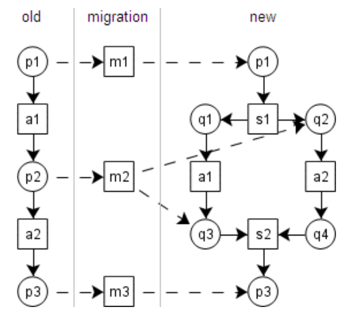
2.2 Operations

In Behaviour Nets, there are some *primitive* operations for modifying the net structure, such as adding and deleting places, transitions, arcs and tokens. Besides the primitive operations, there is a set of more *advanced* operations, which also preserve local soundness. By preserving local soundness we mean that after applying the operation, an execution of the behaviour will still terminate properly, if the behaviour also terminated properly before the operation. The message places Pm are not taken into account when determining local soundness. Local soundness refers to a sound behaviour, to make the distinction with a sound interaction, which will be referred to as global soundness. More information about soundness can be found in [2]. The used set of advanced operations are:

- `division` and `aggregation`*, which divides one transition into two sequential transitions, and vice versa,
- `parallelization` and `sequentialization`*, which puts two sequential transitions in parallel, and vice versa,
- `specialization` and `generalization`, which divides one transition into two mutual exclusive specializations, and vice versa,
- `iteration` and `noIteration`, which replaces a transition with an iteration over a transition, and vice versa,
- `receiveMessage` and `notReceiveMessage`, which adds or deletes an incoming message place,
- `sendMessage` and `notSendMessage`, which adds or deletes an outgoing message place.

For some of the operations, marked with *, is it not always clear how they can be applied on-the-fly, because of the dynamic change problem [3]. For example, `sequentialization`, as mentioned above, cannot be applied for every token marking, as it is not always clear on which places the tokens from the old behaviour should be placed, when migrating to the new behaviour. For modeling the migrations the approach of Ellis et al. [6] is used. By modeling a behaviour change as a Petri net, it can be exactly defined how to migrate the tokens from the old behaviour to the new behaviour. Note that advanced operations can also be described using the primitive operations. For the `receiveMessage`, `notReceiveMessage`, `sendMessage` and `notSendMessage`, nothing needs to be migrated, as there is no change in the places, except for the message place, which initially don't contain a token. In figure B.1 on the next page can be seen how the migration for the operation `parallelization` can be modelled.

Figure B.1: Migration of old to new behaviour



3 Aligning Behaviours

Before two agents start an interaction, they will both individually choose a behaviour they are going to execute, based on what they are expecting of the interaction. An interaction however will not terminate, if the behaviours of the agents interacting are not matching. To overcome this problem, agents are able to change their behaviour on-the-fly, i.e. during the interaction. Alignment policies are used by agents to change their behaviour on-the-fly.

3.1 Alignment policies

An alignment policy is a set of primitive or advanced operations. In our approach, an agent has a set of policies in his knowledge-base from which she can choose when an interaction for example has deadlocked, i.e. when there is no progression anymore in the execution of the behaviour. How an agent will choose an alignment policy (or if she will choose one at all) depends on different factors. The factors discussed next are: kind of problem, beliefs about the agent interacting with, and the willingness to change it's own behaviour.

Kind of problem Most of the time, a problem will occur, when the agent is not receiving the message she is expecting. It can be that the agent did not receive a message at all, or received a different type of message than expected. If she did receive a message, the type of the received message and other factors of the kind of the problem can be used as attributes for selecting the proper alignment policy.

Beliefs about the agent interacting with Beliefs about the other agent can be of great importance when choosing an alignment policy. When for example the agent completely trusts the other agent, she might be willing to make more "sacrifices" in changing her behaviour than when she distrusts the other agent.

Willingness to change behaviour When an agent has very advanced and fine-tuned behaviours, it is not smart to radically change the behaviours because of one exceptional interaction. On the other hand, when the behaviour of the agent is still very primitive, changing it a lot could be a good thing to do. So when an agent gets “older”, and the behaviours are based on more experience, the willingness to change her behaviour will decrease. This approach can be compared with the way humans learn, or with the decreasing of the learning rate over time when training a neural network.

As all this still needs research for what is the best way is to make the decision, a possibility would be to use a heuristic (for example machine learning techniques like a decision tree, neural network or a genetic algorithm), build on experiences of the use of the alignment policies in previous interactions. A new agent won't have any alignment policies, or experience applying them. When an agent does not know how to handle a certain problem, it can go into escape mode, to learn new ways to overcome her lack of experience. More information about the concept of escape mode can be found in [10].

3.2 Example - Proof of concept

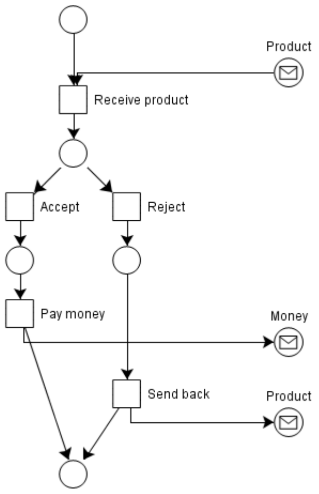
As an example how these alignment policies could work, we give a small example, as a proof of concept. In this example, as shown in figure B.2, the buyer and the seller already agreed on the product the buyer wants to buy, but as seen in the figure, they have different ideas of how the delivery and the payment should go. For the sake of the example, we assume that the behaviour of the buyer is very advanced, and thus has no willingness to change her behaviour. On the other side, the seller's behaviour is still primitive and unexperienced, so we are looking at the problem how the seller can align her behaviour with the buyer, assuming that the seller has trust in the buyer.

When the interaction starts, it immediately deadlocks; the buyer is waiting for the product, and the seller is waiting for the money. A way to overcome this problem would be for the seller to send the product and wait for the money in parallel. So, by using an alignment policy based on the operation `parallelization` the behaviour of the seller changes to the behaviour as seen in figure B.3 (a), and the interaction can continue. However, if the buyer rejects the product, and sends it back, the seller still doesn't have the appropriate behaviour to handle this, because the seller is waiting for the money. In case the seller receives the product back, but when she is expecting the money, the seller could use an alignment policy based on the operation `specialization` to overcome this problem, which divides the receive money transition into two separate transitions, *receive money* and *receive product*. The resulting behaviour can be seen in figure B.3 (b). The behaviours of the buyer (figure B.2 (a)) and the behaviour of the seller (figure B.3 (b)) are now aligned, and thus matching.

An extension of this example showing a more complex interaction concerning a typical business situation is shown in [13].

Figure B.2: Behaviours of buyer and seller

(a) Buyer



(b) Seller

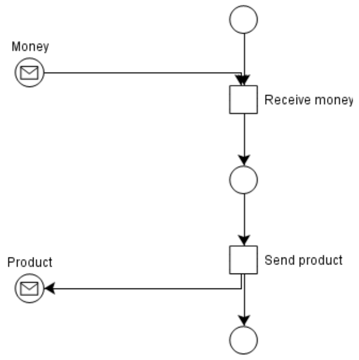
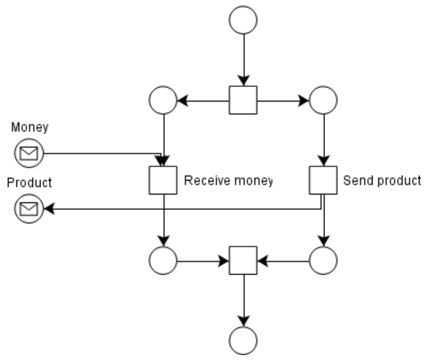
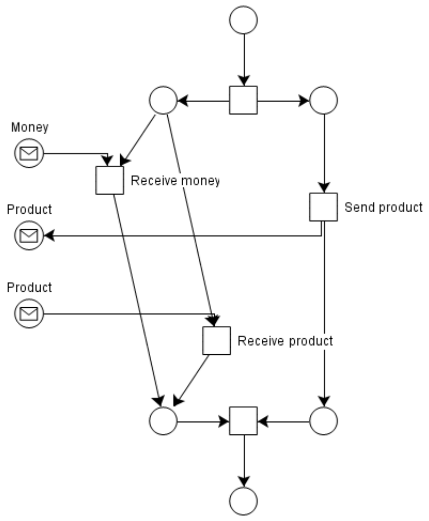


Figure B.3: Adapted behaviour of seller

(a) First adaptation



(b) Second adaptation



4 Discussion and Future Work

This research is preliminary. As far as we know, it is the first attempt to apply this kind of discrete mathematics to anticipatory agents. Although there are some approaches that apply Petri Nets to model agent interaction, these are mainly concerned with a centralistic view. However, we are taking a distributed approach. This approach has the potential to appeal to two research communities: the one oriented towards Business Information Systems development (who apply Petri Net like modelling to BPM and ERP), and also to the growing anticipatory agent community. Some researchers have pointed out that the models used for BIS analysis and design are in fact executable models of the organisation they support. Apparently, the inclusion of a executable model of the organisation in the organisation itself (seen as a system), makes the whole an anticipatory system. Obviously, organisations that use a BIS increase their anticipatory ability. Unfortunately, there is no evidence that the current development of BISs is done with explicit anticipatory ability in mind.

Our strong belief is that agent-oriented BIS that support the business processes of the organisation (in terms of interaction support), due to the anticipatory ability of the individual agents, lead to an emergent behaviour of the whole system that has a anticipatory nature. Of course, such a statement has to be proven empirically and theoretically. An intuition is that simulation - currently intended for development purposes - can have an important role in the anticipatory architecture of an agent-enabled BIS in an organisation. If the executable agent-based model of the organisation can perform simulations itself that start from the present state as perceived in the organisation, this model can predict future states. The results of these predictive simulations can be used to influence via an effector sub-system the current state of the organisation.

Currently, the idea about development simulations is that these are in fact games, where expert players interact with the simulated agents, via the escape/intervention mechanism. An escape is triggered when an agent cannot perform a certain act, and an intervention is when the human supervisor decides that the course of action is not proper. After the agents are fully developed and are deployed in the organisation, the predictive simulations that they could perform should be as automatic as possible (otherwise human intervention would make this anticipatory mechanism inefficient). This observation makes the need for better automatic alignment mechanisms very relevant.

Our future research will be directed towards a number of issues. First, mechanisms for triggering of the escape mode should be investigated, but also what the human will do after the escape is activated, i.e. how to train the agents. Other ways for alignment will also be investigated, like a priori alignment, which can be realised by a superior agent, or even by the agents themselves through a special kind of "pre-alignment interaction" - that would entail negotiation. Superior (software and human) agents can also intervene for alignment on-the-fly (a special kind of escape), and align the behaviour from one agent perspective, or impose a central solution from an global interaction perspective.

Second, for successful enactment of the alignment mechanisms, the way agents can choose an alignment policy has to be investigated. This raises a number of interesting questions. First of all, is an alignment policy a belief? Most likely it is not, as an alignment policy does not contain information about the environment. However, the information used for choosing a policy is based on beliefs about the environment, thus the decision mechanism is probably a belief. Another question is if agents can exchange alignment policies and how she chooses beliefs, and in this way learn from each other new ways of alignment. Such exchanges are regulated in agent societies by trust mechanisms, which means that an explicit representation of trust is needed. Finally, it is needed to figure out how agents can adapt their beliefs about the use of alignment policies.

5 Conclusion

As we have shown, it is possible to describe a policy for alignment that can be applied when the interaction beliefs of two or more interacting agents are not matching. We introduced an extension of Petri Nets to capture the interaction beliefs and also a mechanism to choose the appropriate policy that adapts the beliefs from one agent perspective. From the anticipatory systems perspective, this research can enable predictive agent model execution (agent-based simulation of organisational models) to be more reliable and necessitate less human intervention in terms of alignment.

We believe that interdisciplinary work between the BIS research and anticipatory agent research can yield lots of “cross-fertilisation” and raise the awareness that BIS enabled organisations are in fact anticipatory systems and also provide test beds for novel anticipatory agent ideas.

Bibliography

[1] W.M.P. van der Aalst. Verification of Workflow Nets. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of Lecture Notes in Computer Science, pages 407-426. Springer-Verlag, Berlin, 1997

[2] W.M.P. van der Aalst, Interorganizational Workflows: An approach based on Message Sequence Charts and Petri Nets, *Systems analysis, modelling, simulation*, vol. 37(3), 335-381, 1999

[3] W.M.P. van der Aalst and S. Jablonski, Dealing with workflow change: identification of issues and solutions, *Computer systems science and engineering*, vol 5, 267-276, 2000

[4] P. Davidsson, E. Astor and B. Ekdahl, A Framework for Autonomous Agents Based on the Concept of Anticipatory Systems, *Proc. of Cybernetics and Systems 1994*, pp. 1427-1431, World Scientific, 1994

[5] B. Ekdahl, Agents as Anticipatory Systems, World Multiconference on Systemics, *Proc. of Cybernetics and Informatics (SCI'2000) and the 6th International Conference on Information Systems Analysis and Synthesis (ISAS'2000)*, Orlando, USA, July 23-26, 2000

[6] C.A. Ellis and K. Keddara, A Workflow Change is a Workflow, *Lecture notes in Computer Science*, vol 1806, 201-217, 2000

[7] R. Hamadi and B. Benatallah, Recovery Nets: Towards Self-Adaptive Workflow Systems, *Lecture notes in computer science*, vol 3306, pp. 439-453, 2004

[8] K. Jensen. An Introduction to the Theoretical Aspects of Coloured Petri Nets. In J.W. de Bakker and W.-P. de Roever, editors, *A Decade of Concurrency, Reflections and Perspectives*, volume 803 of Lecture Notes in Computer Science. Springer-Verlag, 1993

[9] C.A. Petri. Kommunikation mit Automaten. PhD thesis, Institut fur instrumentelle Mathematik, Bonn, 1962

[10] Gijs B. Roest, Nick B. Szirbik, Intervention and Escape Mode, *Proc. of AOSE workshop, AAMAS'06 Conference*, Hakodate, Japan, to be published in LNCS, http://tbk15.fwn.rug.nl/tal/downloadfile.php?id=31

[11] R. Rosen, Anticipatory Systems. New York: Pergamon, 1985

[12] K. Salimifard and M. Wright, Petri net-based modelling of workflow systems: An overview, *European journal of operational research*, vol 134(3), 664-678, 2001, Salimifardto appear, 2006

[13] Cees de Snoo, Marco Stuit, Nick Szirbik, Interaction beliefs: a way to understand emergent organisational behaviour, to be published, http://tbk15.fwn.rug.nl/tal/downloadfile.php?id=54

[14] http://tbk15.fwn.rug.nl/