Using Agent Technology to Build a Real-World Training Application

Michal Cap, Annerieke Heuvelink, Karel van den Bosch, Willem van Doesburg
TNO Defense, Security, and Safety
P.O. Box 23, 3769 ZG Soesterberg, the Netherlands.
Phone: +31 346 356 430
Fax: +31 346 353 977
{michal.cap, annerieke.heuvelink, karel.vandenbosch,willem.vandoesburg}@tno.nl

Abstract. Using staff personnel for playing roles in simulation-based training (e.g. team mates, adversaries) elevates costs, and imposes organizational constraints on delivery of training. One solution to this problem is to use intelligent software agents that play the required roles autonomously. BDI modeling is considered fruitful for developing such agents, but have been investigated typically in toy-worlds only. We present the use of BDI agents in training a complex real-world task: on-board fire fighting. In a desktop simulation, the trainee controls the virtual character of the commanding officer. BDI-agents are developed to generate the behavior of all other officers involved. Additionally, agents are implemented to manage the information flow between the agents and the simulation, to control the scenario, and to tutor the trainee. In this paper we describe the design of the application, the functional and technical requirements, and our experiences during implementation.

Categories and Subject Descriptors: I.2.0 [Artificial Intelligence]: General – *Cognitive simulation*; I.2.1 [Artificial Intelligence]: Applications and Expert Systems; I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence – *Intelligent agents, Multiagent systems*; I.6.3. [Simulation and Modeling]: applications; J.7. [Computers in Other Systems]: Military.

General Terms: Design, Human Factors

Keywords: Intelligent Agents, Virtual Training, Multi-Agent System, BDI, Jadex

1 Introduction

Scenario-based simulator training is considered very appropriate for learning decision making in complex environments [1]. A simulation enables trainees to experience the causal relations between actions, events and outcomes in the simulated task environment. It is common practice in simulation-training that Subject Matter Experts (SMEs) (usually staff members) play the role of key players. However, the need for SMEs elevates costs of training, and staff tends to be scarcely available. As a result, there are often too few opportunities for training to achieve the required level of

competence. Organizations are therefore looking for more flexible forms of simulation-training that require fewer organizational and logistic efforts. One solution to reduce the need for staff is to use intelligent software to play the required roles.

Finite State Machines (FSM) are the traditional approach for developing characters acting as key players. In this approach, characters are controlled by defining list of rules and contingencies. FSM has proven its value for designing autonomous characters performing procedural and other constrained tasks. However, for 'open' tasks like tactical decision making, it is hard or even impossible to create a 'spanning set' decision matrix specifying appropriate behavior for all entities for all possible states that may occur during a scenario [2], as in even relatively simple scenarios the number of states tends to be very high [3]. A more promising approach is to develop agents whose behavior is a function of Beliefs, Desires, and Intentions (BDI) [4]. A BDI model represents the knowledge (both generic and situational) and the reasoning processes of an individual or entity in a certain domain, task or scenario. There is a growing conviction that BDI modeling is promising for developing agents being able to handle complex tasks [5]. A considerable amount of research effort has been spent on designing frameworks, tools and specialized programming languages that facilitate implementation of intelligent agents based on BDI [6]. However, application of BDI agents have so far typically been demonstrated in toy-world environments [7,8,9].

In this paper we present the design, development, and implementation of a scenario-based simulator training of a complex real-world task in which BDI agents act as virtual team members. Our aim here is to describe the design of the application, to account for our choices to fulfill the identified functional and technical requirements, and to report our experiences during implementation. In the next section we will briefly introduce the domain and the purpose of the training application.

2 Training Application

For the Netherlands Defence Organization we developed an agent-based desktop training simulation. The domain is on-board fire fighting, and the task to be trained is that of the commanding officer, the Officer of the Watch (OW). If a fire breaks out, the OW is in charge of handling the incident. Operating in the Machinery Control Room (MCR) of the ship, he contacts his team, develops a plan to contend the incident, gives orders, monitors the events, and adjusts plans if necessary. The OW communicates with four other officers: Chief of the Watch (CW), MCR-Operator1 (MCRO1), Leader Containment Party (LCP), and the Leader Main Party (LMP). The first two are for the greatest part also situated in the MCR, the last two are at or near the incident scene.

The task of the OW is a typical example of decision making in a complex environment. There are, of course, procedures for handling a fire accident. However, the OW also has to anticipate on possible complications, needs to respond to unforeseen actions, has to adjust plans when events require him to do so, and so on. In the same manner, his team members need to be able to react to unforeseen events and new actions of the OW.

In the desktop-simulation training, the trainee controls the virtual character of the OW. For an impression of the training simulation, see Figure 1. In addition to the OW, the aforementioned team members are also represented as virtual characters in the simulation. In order to achieve good training, the trainee must be able to practice the task in accordance with the way an OW does this in the MCR. This means, for instance, that all relevant information and communication equipment needs to be simulated, and team members needs to respond in a realistic fashion to the trainee and to each other. However, the application is not mere simulation, it concerns a *training* simulation. This implies that the scenario needs to be managed in order to make sure that the emerging situations support achievement of the learning objectives. Furthermore, the quality of the trainee's performance must be evaluated and appropriate feedback needs to be given. In the next section we elaborate on the functional requirements to ensure successful desktop-simulation training.

Fig. 1: Impression of the desktop simulation training

3 Functional Requirements

In previous research we identified several functional requirements the training simulation needs to adhere to [10, 11]. In this section we describe and extend these functional requirements. Some requirements follow from choices concerning the scope of the training, e.g., only the OW is trained, others from the motivation to implement a full-fledged stand-alone training system, which includes believable behavior of virtual team members and a tutoring functionality.

1. To ensure transfer of training, the trainee's task environment needs to be simulated in a functionally realistic way. During fire fighting, the Officer of the Watch (OW) is situated in the machinery control room (MCR) of the ship, while several team members enter and leave the MCR. So in the training system several virtual characters representing the trainee and the team members are required, as well as a realistic simulation of the MCR.

2. A great part of the OW training concerns the interaction with team members. To train this realistically, it is required that the OW can interact with the other virtual characters with some degrees of freedom. On the other hand, to keep the training scenario under control, communication should be regulated.

3. The behavior of the virtual characters has to represent that of task experts, which entails that they should be able to autonomously perform their task. Moreover, they should be able to reason about their task and, e.g., propose an alternative plan if the trainee proposes a wrong one.

4. To execute their task, the virtual characters need to have knowledge and reason about what has happened in the past. E.g. the Chief of the Watch (CW) should decide to plot the fire location on the damage control board if the fire broke out more than 30 seconds ago and if it has not yet been plotted by anybody else.

5. Some team members need to perform tasks in other parts of the ship than the MCR. In addition, the trainee needs to receive information coming from outside the MCR. So besides the visualized model of the MCR, we need a functionality that models events taking place in the rest of the ship (which does not need to be visualized).

6. The team members that enter and leave the MCR during fire-fighting will execute tasks in the MCR, as well as on the rest of the ship. For them it should not make a difference whether their task environment is visualized (the MCR) or modeled by another functionality (the rest of the ship, see 5).

7. To ensure believable behavior, what a team member can perceive should be determined by its location. For example, initially only team members present in the MCR should know the status of the incident, based on the damage-control board. However, once the status has been broadcasted by the OW, everybody knows it, because broadcasts are audible at each location.

8. A stand-alone training system requires a tutoring functionality. For tutoring it is foremost required to evaluate the trainee's behavior. For this the system should be able to compare trainee actions with a specification of expert actions. Moreover, this comparison should lead to an appropriate tutoring action.

9. Because goal-directed, systematic training is more effective than learning by doing, we want to implement a scenario-based training system [12]. In order to control the training scenario, we need a functionality that (or inhibits) events at appropriate times during the scenario.

10. To function coherently, the virtual characters and other functionalities of the training system require up to date information about the state of the simulated environment and the actions of the trainee. Unfortunately, a simulated environment will generate so much data that, if unmanaged, this information exchange will slow down the training system. We therefore need a functionality to channel this data, ensuring that only relevant information is transferred and without significant delays.

11. Our current focus is the development of a prototype training system, with the intention to develop it further for operational use in naval staff training. We therefore require our training system to be based on a stable software platform.

12. Future operational use of the program requires that the current prototype containing one example scenario should be easily extendable with new scenarios. This makes it necessary to use a programming environment that is transparent, expressive, and easy to work with.

4 Technical Solutions

In this section we shortly discuss the technical solutions we selected for our training system for each of the functional requirements introduced above.

1. To develop a visually realistic representation of the environment, we decided for a 3D representation of the MCR generated by a 3D game engine developed by our industrial partner "VSTEP"[1]. All equipment that is normally used by the OW is simulated and available to the trainee (damage control board, information panels, communication equipment, etc). In addition five virtual characters are modeled, one representing the OW, the others representing his team members.

2. In reality, team members communicate by speech. However, using speech in this training application would introduce at least two major problems. First, speech recognition technology is not yet advanced enough to recognize and interpret spoken messages, certainly if the syntax and vocabulary is unrestricted and the system is untrained to pronunciation characteristics of the speaker. A second problem is that natural speech would introduce so much freedom that it would be very difficult to control the scenario. As a solution, we decided to allow the trainee to communicate with its team members using pre-established context-sensitive menus that are dynamically filled with communication acts based on the current state of the training simulation, see Figure 1. All the OW's possible communication acts, as those of his team members, are pre-recorded using a speech synthesizer. For an extensive discussion on the ontology-driven dialog system underlying this feature of our training system, see [13].

3. To develop virtual characters that act as experts, domain knowledge is required. Because experts tend to explain their actions in terms of beliefs, goals and intentions, expert knowledge can be easily translated to a BDI model [14]. It has been demonstrated that software agents based on the BDI-paradigm can provide virtual characters with believable behavior in computer games [15], and in virtual training [16]. BDI agents commonly incorporate a plan-base that embeds the information on how to reach specific goals. Decision making in fire fighting is often procedural in nature: plans for achieving goals under given conditions are thus available. Because of all these benefits, we decided to implement software agents based on the BDI paradigm to control the non-player virtual characters. We refer to these agents as the *role-playing agents*.

---

1 http://www.vstep.nl

4. Due to the notion of beliefs, the agents have knowledge about the current situation. However, having beliefs does not necessarily entail that the agent can remember what was previously the case. To allow for temporal reasoning, we decided to store time annotated beliefs about every change of the world state into the agents' historical belief bases.

5. In order to model and simulate parts of the ship that are not visualized in the 3D simulation, we introduce a **world manager agent**, which is one of the implemented *functional agents* as opposed to the *role-playing agents*. In contrast to a role-playing agent, a functional agent does not represent an individual in the scenario, but instead fulfills one or more functions on the background needed to manage the simulation-based training. The world manager agent does not only simulate events outside the MCR, it also simulates the actions (with time and effects) of role-playing agents not present in the MCR.

6. To ensure that the role-playing agents can interact in the same way with the visualized (MCR) as with the non-visualized part of the simulated environment (rest of the ship), we decided to channel all actions and observations through the **world manager agent**. The fact that this functional agent maintains the complete world state of the entire simulation also ensures that no inconsistencies arise between the visualized and non-visualized part.

7. Because the world manager agent is informed about the exact state of the world, it knows the location of each role-playing agent, and thus what that team member can see and hear. Below (see 10) we explain how this information is used to channel the information flow between the role-playing agents and the 3D simulation.

8. In order to keep track of the actions that the trainee executes in the simulated environment, to evaluate the trainee's performance, and to decide upon appropriate interventions (if any), we introduce an **OW/Tutor agent**. This agent is interesting in the sense that it is a functional agent, embedding a role-playing agent. The BDI task knowledge underlying the actions of role-playing agents can be reused in the tutor agent to evaluate behavior of the trainee. If a trainee's action coheres with the task model (in this case of the OW), it is evaluated as correct. If the trainee fails to do what is required according to the task model (e.g. "hail the fire within 30 seconds after a positive fire alarm"), the OW/Tutor Agent interprets this as an error (constraint broken). In addition, the OW/Tutor agent has a function deciding whether and which intervention to take (e.g. letting the Chief of the Watch give a reminder).

9. For starting the training scenario and keeping it on track by triggering events in the simulated environment we use a **scenario manager agent**. This agent triggers events based on a pre-defined scenario model (e.g. a fire of type A starts at location X). By separating this knowledge in a specific functional agent, we intend to facilitate the future extension of the training simulation with additional scenarios (see 12).

10. In order to keep the data flow between agents and the 3D simulation manageable, we decided that agents only receive information that is relevant to them. We achieve this by using a publish/subscribe paradigm in which information producers and consumers coordinate among each other what information to exchange. We introduce a functional agent called the **broker** to carry out this task. Publish/subscribe mechanisms are provided by common middleware standards like the Data Distribution Service (DDS), Java Messaging Service (JMS) and High Level Architecture (HLA). However, these mechanisms define static contracts, whereas our

agents' information need is dynamic (e.g. information exchange is, for instance, dependent upon the agent's location in the ship). We introduce a dynamic filtering system that adjusts the subscriptions of the agent to information supplied by the simulation in such a fashion that it corresponds to its specific situation or context (the knowledge for this is provided by the world manager agent, see 7). In this way, the agent only receives information that it potentially can receive.

11,12. To be able to implement the system cost-efficiently, we looked for an existing BDI-based agent platform. Since the prototype under development will at a later stage be further developed for operational use in naval staff training, the stability and performance of the selected platform is of major importance. We have considered three agent platforms to implement the role-playing and functional agents: JACK, Jadex, and 2APL.

JACK is a mature, proven agent platform providing commercial support. However, we experienced the provided development environment as rather unintuitive and the predefined workflow too inflexible for our needs.

Jadex is an academic project aiming to develop an industrial strength agent platform. It has been shown to be applicable in a real-world medical planning application [17], which made us believe that the performance of the system would be sufficient for our system. Agents in Jadex are specified using a relatively complex and verbose XML-based language. Since agents in Jadex can manipulate arbitrary Java objects and the platform is fully open-source, we considered the provided framework extensible enough to allow implementation of our specific needs.

2APL is an academic, research-oriented agent programming language adhering more than the other platforms to the theoretical principles of BDI. However, the current implementation has not been designed for computational efficiency and it has not been tested yet in operational contexts.

The main criteria for selecting an agent platform were the stability of the platform, its performance, and the ability to customize the provided framework. In particular, we required the platform to support incorporation of historical belief bases and a publish/subscribe mechanism for messaging. We decided to drop 2APL because it does not provide production-level stability and performance. Next, after comparing JACK and Jadex, we decided to choose Jadex because of its extensible design and its open source code, as this would allow us to attach our own functionalities.

5 System Architecture

Following the functional requirements and technical solutions stated above, we implemented a training system that consists of two major parts: 1) a 3D visualization that serves as an interface to the simulation for the trainee, and 2) an agent system accommodating a) role-playing agents that model behavior of the virtual characters and b) functional agents that manage the simulation, the execution of the scenario, and tutoring. For an overview of the training system, see Figure 2. In this section we will elaborate on the functionality of each part of the training system.

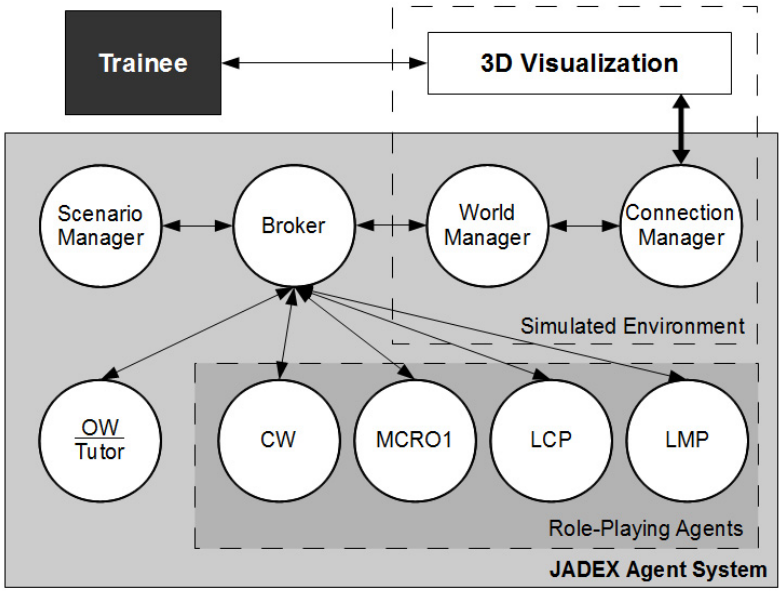

Fig. 2: Diagram of the system architecture

3D Visualization

Once a training session starts, the trainee can control his virtual character in the 3D game environment using the keyboard and mouse (section 4.1). Each interaction with the simulated environment (such as: picking up the headset, leaving the MCR, talking to another character) generates a *cue (*a data structure that describes the change of the world state), which is then relayed to the agent system. Conversely, the agent system can send a variety of commands to the 3D engine in order to force some of the entities to perform a certain action. E.g. the CW role-playing agent may order the virtual character of CW to sit on his chair; the virtual character immediately starts moving towards the chair and once there, he plays the animation of the sitting move; after the successful completion of the action, the system generates a cue "the chair of CW is taken", which is then forwarded to all the virtual characters (and subsequently their attached role-playing agents) currently present in the room. Further, the 3D engine provides a dialog interface that allows a trainee to communicate with his team members (section 4.2). Using the interface, the trainee can choose from a wide variety of sentences that can be communicated by clicking on one of them; the engine plays the corresponding pre-recorded speech and generates a communication cue which is subsequently forwarded to the interlocutor. Since all communication between two virtual characters should be audible by the trainee (if he is in the same room), the role-playing agents communicate through their virtual characters in the simulation. After

the appropriate speech file is played, a communication cue is relayed to the corresponding receiver and other characters that may have overheard the communication (section 4.7).

Connection Manager – *functional agent*

When the **Connection Manager** is started it establishes a connection to the 3D visualization and starts all the other Jadex agents. The connection manager serves as the gateway between the 3D visualization engine and the agent system. Being the connection gateway between the simulation and the agent system the connection manager has one additional task: translating the cues originating from the 3D visualization into a format understandable to the agent system, and vice versa, translating the actions coming from the agent system into the language of the 3D visualization. These translations are made based on a dedicated mapping of the ontology used within the 3D simulation to the ontology used within the agent system.

World Manager – *functional agent*

Since the virtual environment is divided into a visualized part (MCR), which is simulated within the 3D visualization, and a non-visualized part (the rest of the ship), the agent system contains a **World Manager** whose responsibility is to simulate the non-visualized part and to control the blending of both (section 4.5). As mentioned in section 4.6, the role-playing agents should not notice any difference between their interaction with the visualized, and non-visualized environment. Therefore all the cues from the visualized environment, denoting changes in the world state, are first send to the world manager that uses them to update its belief base containing the overall world state, ensuring that no inconsistencies exist. Next, these belief updates are forwarded to the rest of the agent system. Similarly, all role-playing agent actions are first send to the world manager that assesses whether or not they take entirely place in the MCR. If this is the case, the action is forwarded to the visualized environment to be executed, resulting in cues from this environment to the world manager, who will forward them to the appropriate agents. If an action takes (partly) place in the non-visualized environment, the world manager executes the action and generates (and forwards) the cues denoting the effects of the action.

To illustrate this function, consider the following example: During fire fighting, the MCR operator (MCRO1) is ordered to check the smoke valves. In reality this involves walking to a location near the fire, checking visually the status of the valves, then returning to the MCR. In our system, this action requires blending of the visualized and non-visualized world and is therefore executed by the world manager. The world manager first instructs the virtual character of the MCRO1 to walk out of the MCR (visually observable to those inside the MCR), and then lets the MCRO1 "wait" around the corner (not observable to those inside the MCR). Based on the real walking distance to the fire location, the world manager calculates how long the MCRO1 needs to wait outside the MCR before it can generate the cue denoting the

result of the visual inspection, forwards this cue to the MCRO1, and instructs the MCRO1 to return into the MCR.

CW, MCRO1, ATL, CTL – *role-playing agents*

The four **Role-playing agents (CW, MCRO1, LCP, LMP)** represent the team members of the OW. The models underlying their behavior consist of expert task knowledge in the form of beliefs, goals and plans (section 4.3). There may be several reasons for a role-playing agent to initiate action. It may originate from his own task knowledge, for example if circumstances in the world form beliefs that result in a goal to become active, after which the agent selects a plan (series of actions) to achieve the active goal. It may also originate from a request for information, or an order, given by the trainee or other role-playing agents. Sometimes beliefs may trigger more than one goal. The role-playing agent then has to decide which goal to pursue based upon priority rules. The agents store all messages concerning changes of the world state (received from the world manager) in their historical belief base (section 4.4). In our application, role-playing agents do not act directly in the simulated world. They do so by sending action messages via the broker to the world manager, who controls their virtual character accordingly. Because the models of role-playing agents are based on beliefs and goals, they are able to respond to questions from the trainee by inspecting their belief base, and can adopt any order as a goal.

OW/Tutor – *functional agent*

The **OW/Tutor** agent embeds two models: one model is an expert model of the OW, the second model is a user model of the trainee. The expert task model is static and established before the training. The trainee user model is dynamic, and formed during training. For example, when the trainee asks about the state of the ventilation and receives an answer, the OW agent stores this action and the received information in its belief base, building up the user model. This information about the user, in combination with the task knowledge embedded in the OW agent, can be used to generate an evaluation of the performance of the trainee (section 4.8). Based on this evaluation, the agent can decide to adjust the scenario in order to achieve more efficient training. This can be done indirectly by ordering the role-playing agents to change their behavior.

Broker – *functional agent*

The **Broker** represents the pivot of the publish-subscribe mechanism described in section 4.10. It contains a database of subscriptions determining which messages will be forwarded to each of the agents. It is the responsibility of the world manager agent to maintain the subscriptions for each agent, ensuring they are consistent with the constraints imposed by the world on what an agent can possibly sense. Consider the following example as an illustration of this mechanism: When the CW enters the

MCR, the world manager subscribes him to all the beliefs related to the entities located in the MCR (status of a phone, availability of a chair etc.) Through the newly established subscriptions, the CW receives messages about any change in MCR. Analogically, as soon as the CW leaves the MCR, the world manager breaks all those subscriptions, after which the CW will not be informed anymore about changes in the MCR.

Scenario Manager – *functional agent*

The **Scenario Manager** agent controls the course of events in the training simulation using a pre-specified scenario definition (section 4.9). The latter denotes, e.g., where a fire should start, what type of fire it is, and whether the ventilation of the ship will, or will not crash stop. Based on the scenario definition and generic knowledge about the environment stored in its own task model, e.g. about how the fire will react to certain events, the scenario manager agent will at appropriate times during training execute specific events like starting a specific type of fire at a specific location, expanding that fire, and crash stopping the ventilation.

6 Implementation

As mentioned in Section 4, we decided to implement the agent-system outlined above using the Jadex platform [9]. Due to space limitations, we will not discuss the implementation in full detail. Instead, we present a few examples of representative issues we had to deal with, and reflect on the suitability of features available in Jadex for implementing our agent-system.

Defining an Ontology and Fixing the Message Format

After analyzing the domain, scenario and tasks, we found that we need about 40 types of beliefs to describe the states of the simulation environment (e.g. location of an agent, status of the fire siren etc.). Each belief has about 2 - 3 parameters. Similarly, we identified 52 actions required for the role-playing agents to perform their task. It is apparent that we need an appropriate ontology to handle the identified actions and belief types efficiently.

The Jadex platform supports use of ontologies created in the Protégé environment through the provided Beanynizer plugin-in, which is able to generate JavaBeans for the modeled ontology. Consequently, each JavaBean may be serialized to XML and can easily be transferred between agents as the content of a FIPA message.

Although the provided framework is very powerful, we considered this approach as too heavy-weight for our needs and decided to define our ontology as a collection of classes that provide static constants representing the domain concepts. That way, we can enjoy facilities such as code auto-completion available in the modern Java IDEs. Moreover, most of the syntax errors are discovered already by the Java editor and

other problems concerning incompatible use of beliefs or actions are reported as soon as the agent code is compiled.

Further, we feared that the XML representation of beliefs might become unpractical when one needs to debug the multi-agent system. Therefore, we have developed a more space-economic serialization format, which is able to carry all the information the agents need to exchange, namely belief updates, actions, behavioral instructions[2] and broker messages[3]. To illustrate the format we used: suppose the CW agent decides to check the status of the fire fighting pumps. Using the methods available in our ontology, he can create an object that represents the action, which may be serialized as follows:

```
(messagetype:action)(source:cw)(type:mcrpanelaction)(id
:checkpumps)(timeout:0)(location:panel_cw)(actiontype:c
heckpumps)
```

This representation proved practical for debugging purposes as one message usually fits to one line in a debugging output. Besides, we designed the message format almost identical to the structure of cues and actions used within the 3D visualization engine, which makes the conversion of data exchanged between the two systems fast and straightforward.

Unfortunately, this simplified message format revealed its limitations in later stages of the project development. In particular, we found it problematic that the format is unable to naturally express nested data structures, such as a query about a belief, which forced us to come up with rather complicated workarounds. We now recognize that the built-in ontology support, most likely, would have served us better.

Historical Belief Base

The domain of fire-fighting requires agents to reason about time in relation to actions and beliefs. For example, if the fire alarm has been active for more than 30 seconds and there is still no attack plan plotted on the damage control board, the tutor agent can decide to remind the trainee of his duties by sending a behavioral instruction to the CW to suggest to the trainee to plot an attack plan. In order to allow for such reasoning over states in time, the agents requires access to a history of beliefs (when was is first believed that there was a fire alarm?) and actions (is there not attack plan plotted since that time?). In addition, a history of the beliefs and actions of each agent can be used to off-line evaluate the trainee's performance.

To our knowledge, Jadex does not offer support for temporal reasoning. In fact none of the considered BDI-platforms do. However, Jadex allows to add a time parameter to each belief and keep all adopted beliefs in a dedicated set. That way an

2 A behavioral instruction is send by the Tutor agent to modify the behavior of a role-playing agent. The goal of intervening in the agent's goal state is to bring about behavior that helps the trainee in achieving a learning objective.

3 Broker messages are used to indicate that a) an agent is capable to generate a certain message b) an agent is interested in messages of a certain type c) the broker has established a subscription between an information producer and an information consumer.

agent has access to the complete history of his beliefs and is capable of deriving time-related conclusions.

A historical belief base could be implemented in two ways, a) by employing the belief set concept available in Jadex or b) as a single Jadex belief that encapsulates (since it is a Java object) all the desired functionality. Since we considered it practical to be able to define custom methods to search and update the beliefs stored in the belief base, we chose for the latter solution. We found that the custom belief base specified in the above mentioned way enabled us to access the historical beliefs both from conditions in the agent definition files and from Java plans in a natural and effortless way.

Logical Inferences

To perform their role as task experts, the role-playing agents need to be able to make logical inferences. For example, if an officer hears that the siren goes off, he needs to be able to derive that there is a fire alarm coming from an unknown source.

The ability to specify implication rules as a part of the belief base of an agent is one of the fundamental features of logic-based agent-oriented languages. However, since the belief base in Jadex is not logic-based, but specified as a collection of Java objects, it does not provide support for logical inferences.

To overcome this limitation, we have implemented a simple custom inference engine, which is executed each time a new belief is added or removed from an agent's belief base. The inference engine is implemented as a Java class with a designated method that contains hard-coded implication rules that generally follow this structure:

```
if <query on belief update>/<query on belief base>
then <update the belief base>/<adopt a goal>
```

Besides its original use to deduce new beliefs from existing beliefs, we found that this system was also useful for generating reactive behavior (i.e. when an incoming belief update makes the agent adopt a new goal) and for decomposing complex beliefs (e.g., when a new attack plan is received, atomic parts of the plan are also added to the belief base). This turned out to be a robust approach: although our inference engine contains tens of inference rules, it did not noticeably decrease system performance.

XML vs. Java

One of the distinguishing features of Jadex is its ability to program agents in both XML and Java. In fact, a programmer may decide whether he prefers to specify the major part of his agent declaratively in an XML-based agent definition file (ADF) or in a more imperative way using Java plans. However, from a theoretical point of view, it is more natural to program agents declaratively [18].

To explain the difference between using XML and Java to implement certain task behavior, consider the following example of desired role-playing agent behavior: When the siren goes off, the MCR operator must check whether the ventilation has been automatically crash stopped, and inform the CW about the result of this check.

Only after that, he can use his work station to check the location of the fire alarm, and turn of the siren.

In this example we began by implementing this behavior in the XML-based agent definition file. For this, we first modeled that the MCR operator agent adopts the goal "deal_with_siren" as soon as it believes that the siren is active. When a goal becomes active, the agent constantly tries to execute all the associated plans. In the case of the "deal_with_siren" goal, multiple plans are associated: one for checking the status of the ventilation, one for telling this status to the CW, another for checking the location of the fire alarm, and yet another for turning the siren off. To ensure the correct execution of his task, the preconditions of these plans needed to be specified in such a way that in fact, they will be performed sequentially. We found that this led to very complex preconditions, e.g.:

```
<precondition>
  <!-- MCRO1 knows the status of the ventilation. -->
  $beliefbase.worldbeliefs.contains(Ventilation
    .getFunctor())
  <!-- It has not been communicated yet. -->
  $beliefbase.communicationbeliefs
    .getLastInformPattern($beliefbase.id, Officers.CW,
    null, Ventilation.getFunctor()) == null
 </precondition>
```

Next we implemented the identical behavior in a single Java plan that existed of a number of if-clauses that ensured that the "deal_with_siren" subtasks would be executed in the correct order . Based on this and further experiences we noticed that, although the XML-based approach fits the agent-orientation notion better, our programming productivity significantly increased when employing the Java-based approach. This was mainly due to the availability of sophisticated productivity features for Java code such as code hints and instant source validation. None of these features are available to support the development of Java-like conditions in Jadex ADF files. Also the debugging of the Java code proved to be easier, as the error messages generated by the Jadex XML parser often provided insufficient details to identify the problem. For the above mentioned reasons, we found it easier to specify most of the agent's logic in the Java plans.

Jadex v. 1 vs. Jadex v. 2

When we started the implementation of our agent-based training system (May 2009), the Jadex agent platform was available in two versions: a) Jadex v. 0.96 b) Jadex v. 2-beta2. As the latter version has undergone a complete re-design, which is expected to bring significant improvements in performance and flexibility [19], we aimed to base our system on Jadex v. 2. However, the problems stemming from the lack of documentation and instability of the platform we were facing in the first days of development made us switch to Jadex v. 0.96. This version has proven to be relatively well-documented and reasonable stable.

7 Conclusion

In this paper we have reported our efforts to develop a stand-alone training system for a complex real-world task based on software agents. We have adopted a BDI-approach to develop role-playing agents that can act autonomously and intelligently, and selected the Jadex agent platform for implementation.

It took us about 80 man-days of implementation to build the agent system for the desktop simulation for training on-board fire-fighting. The agent system consists of 9 agents, 359 Java classes and 11 ADF files defined by 17 thousands lines of Java code and 2.5 thousands lines of XML code. The system satisfies our criterion that it should be able to react to events taking place in the simulation without any noticeable delays.

Although the coverage and quality of the programming and debugging tools bundled with Jadex are not comparable to mainstream development environments (e.g. Java + Eclipse), the Jadex agent framework proved to be a robust and well performing platform. We appreciated in particular that Jadex does not force programmers to work with the built-in notions of the BDI-components and allows users to extend or redefine them. For example, Jadex allowed us to bypass the platform's default belief base and switch to a custom-made historical belief base.

The work presented in this paper covers the implementation of the role-playing agents and their interaction with the 3D visualization. For the former the focus lay on the realization of valid expert behavior in a wide variety of emerging situations. For the latter we focused on an efficient, reusable manner to link BDI role-playing agents to a virtual simulation, for which we developed several specific functional agents.

Although we have far-stretching ideas on how to model the tutor agent conceptually [10, 11], we have as yet not been able to invest much efforts into implementation and integration of these ideas in the introduced OW/Tutor agent. Although all the functionality is there for the OW/Tutor agent to record and reason about the behavior of the trainee, and to order other agents (role-playing ones, but also the scenario manager agent) to perform certain actions (which they can also already execute), we have not spend time on defining the rules to govern which actions should be ordered to support the trainee in reaching his training objectives. In future work we will extend the knowledge of the tutor agent with such rules of intervention to exploit the innovative opportunities offered by BDI agents to full extent: on-line control of agent behavior, ensuring it is supportive to the training goals. A short example may illustrate this point. Suppose that a tutor assesses that a trainee is not challenged to learn by the events when presented with a standard scenario. The tutor agent may decide to introduce events that do challenge the trainee in a useful way. He may for example send a behavioral instruction to the CW-agent prompting it to "forget" switching off the electricity at and near the incident scene. This forced event enables the trainee to achieve the learning objective of checking whether all safety precautions have been taken and to make corrections, if necessary. Such on-line control of autonomous training is the wish of many, and seem to come within reach by linking BDI agents to training simulations.

Acknowledgement
This research project (032.13359) has been funded and supported by the Netherlands Department of Defence.

References

1. Oser, R.L. (1999). A structured approach for scenario-based training. In Proceedings of the 43rd Annual meeting of the Human Factors and Ergonomics Soc. Houston, TX, 1138-1142.
2. Silverman, B.A. (2001). More realistic human behavior models for agents in virtual worlds: emotion, stress and value ontologies. (Report No. Technical Report). Philadelphia, PA: Univ. of Penn/ACASA.
3. Klein, G. (1998). The source of power: how people make decisions. Cambridge, MA: MIT.
4. Bratman, M. E. (1987). Intentions, Plans, and Practical Reason. Harvard University Press, Cambridge, MA, USA.
5. Pew, R. W. and Mavor, A. S., editors (1998). Modeling human and organizational behavior: Application to military simulations. National Academy Press, Washington, DC, USA.
6. Georgeff, M. P., Pell, B., Pollack, M. E., Tambe, M., and Wooldridge, M. (1999). The Belief-Desire-Intention Model of Agency. In Proceedings of the 5th international Workshop on intelligent Agents V, Agent theories, Architectures, and Languages. Lecture Notes In Computer Science, vol. 1555. Springer-Verlag, London, 1-10.
7. Astefanoaei, L., Mol, C., Sindlar, M., & Tinnemeier, N. (2008). Going for Gold with 2APL. In Proceedings of the fifth international workshop on programming multi-agent systems. Springer.
8. Bordini, R. H., Wooldridge, M., & Hübner, J. F. (2007). Programming multi-agent systems in agentspeak using Jason, p.180 (Wiley Series in Agent Technology). John Wiley & Sons.,
9. Pokahr, A., Braubach, L. Lamersdorf, W. (2005). Jadex: A BDI Reasoning Engine. In Multi-Agent Programming, R. Bordini, M. Dastani, J. Dix and A. Seghrouchni, Eds. Springer Science+Business Media Inc., USA, 149-174.
10. Bosch, K., Harbers, M., Heuvelink, A., and Doesburg, W. A. (2009). Intelligent Agents for Training On-Board Fire Fighting. In Proceedings of the 2nd international Conference on Digital Human Modeling: Held As Part of HCI international 2009. V. G. Duffy, Ed. Lecture Notes In Computer Science, vol. 5620. Springer-Verlag, Berlin, Heidelberg, 463-472.
11. Heuvelink, A., Bosch, K. van den, Doesburg, W. A, & Harbers, M. (2009). Intelligent Agent Supported Training in Virtual Simulations. In Proceedings of the NATO HFM-169 Workshop on Human Dimensions in Embedded Virtual Simulation. Orlando, FL: NATO Human Factors and Medicine Panel.
12. Blackmon, M.H., & Polson, P.G. (2002). Combining Two Technologies to Improve Aviation Training Design. In Proceedings of Human Computer Interaction (HCI). AAAI Press, 24-29.
13. Oijen, J., van Doesburg, W.A., & Dignum, F. Goal-based Communication using BDI Agents as Virtual Humans in Training: An Ontology Driven Dialogue System (submitted).
14. Norling, E. J. (2004). Folk psychology for human modelling: Extending the BDI paradigm. In Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS) Washington, DC: IEEE Computer Society, 202-209.
15. Norling, E. (2003). Capturing the Quake Player: Using a BDI Agent to Model Human Behaviour. In Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS). Melbourne, Australia, 1080-1081.
16. Bosch, K. van den, & Doesburg, W.A. van. (2005). Training Tactical Decision Making Using Cognitive Models. In Proceedings of the Seventh International NDM Conference. Amsterdam, the Netherlands.
17. Paulussen, T. O., Zöller, A., Heinzl, A., Braubach, L., Pokahr, A., and Lamersdorf, W. 2004. Patient scheduling under uncertainty. In Proceedings of the 2004 ACM Symposium on Applied Computing (SAC '04). ACM, New York, NY, 309-310.
18. Shoham, Y. (1993). Agent-oriented programming. Artificial Intelligence, 60(1), 51-92.
19. Pokahr, A., Braubach, L. (2009) From a Research to an Industrial-Strength Agent Platform: Jadex V2 in: 9. Internationale Tagung Wirtschaftsinformatik.