A Methodology for Developing Self-Explaining Agents for Virtual Training

Maaike Harbers[1,2], Karel van den Bosch[1] and John-Jules Meyer[2]

[1]TNO Human Factors, P.O.Box 23, 3769 ZG Soesterberg, The Netherlands
[2]Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
{maaike,jj}@cs.uu.nl, karel.vandenbosch@tno.nl

***Abstract*—Intelligent agents are used to generate the behavior of characters in virtual training systems. To increase trainees' insight in played training sessions, agents can be equipped with capabilities to explain the reasons for their actions. By using an agent programming language in which declarative aspects of an agent's reasoning process are explicitly represented, explanations revealing the underlying motivations for agents' actions can be obtained. In this paper, a methodology for developing self-explaining agents in virtual training systems is proposed, resulting in agents that can explain their actions in terms of beliefs and goals.**

I. Introduction

Virtual training systems are often used to train people for complex, dynamic tasks in which fast decision making is required, e.g. the persons in command in crisis management, military missions or fire-fighting. During a training session, trainees interact with other virtual players, such as team-members, opponents, or colleagues from other domains. Using intelligent agents to generate the behavior of these virtual players lets trainees train at any place and time, reducing costs.

Typical mistakes of trainees include giving incomplete or unclear instructions, forgetting to monitor task execution, and failing to pick up new information and quickly adapt to it. Many of these errors involve situations in which a trainee makes false assumptions about other agents' knowledge or intentions. For example, a commanding fire-fighter who is in a fire truck to contact the dispatch center will not hear that one of his team members yelled that he saw a victim. His team members, however, might not have seen the commander in the truck and unjustly assume that he heard the message, leading to suboptimal behavior. Evidence for the origin of such mistakes can be found in literature; attributing incorrect knowledge and intentions to others is a well described phenomenon in cognitive sciences. For example, Nickerson gives an overview on literature about the tendency to ascribe one's own knowledge to others [11], and Keysar reports on limits in theory of mind use in practice, i.e. attributing incorrect mental states to others [6].

To improve trainees' performances, they should be made aware of their (possibly) false assumptions about others. Better understanding of past sessions should make trainees more alert and decrease the probability that they will make similar errors in a next incident. Therefore, we propose the use of self-explaining agents in virtual training, i.e. agents able to explain the reasons for their actions. Humans usually explain and understand their own and others' behavior in terms of beliefs, desires and other mental contents [5]. Therefore, to provide useful explanations for humans, agents should explain their behavior in a similar terminology, e.g. by revealing the goals they held during a training session. Such explanations serve to increase the trainees' awareness of other agents' perspectives.

Current approaches of explanation in artificial intelligence do not provide explanations from an intentional perspective, that is, in terms of goals. Expert system explanations usually provide traces of the steps behind an diagnose or advice, and the justifications of those steps [17]. However, behavior of expert system is usually not understood in terms of intentions, in contrast to behavior of virtual characters. There are a few accounts of self-explaining agents in virtual training systems [8], [16], [3], but these do not provide information about the actual goals behind an agent's actions.

In order to explain agent behavior in terms of intentions, agents must be implemented in such a way that they act on the basis of intentions, and that their intentions are explicitly represented and thus available for the generation of explanations. In other words, behavior generation and explanation are connected; the reasoning steps that generate an action, are also used to explain that action. Making a connection between generation and explanation of behavior can be achieved by implementing self-explaining agents in a BDI-based agent programming language, because in those languages the declarative concepts needed for explanation are explicitly represented.

In this paper we introduce a methodology for developing self-explaining agents for virtual training systems. The different steps in the method are: determining the required scope of an agent's behavior (section II), constructing a task hierarchy of the agent (section III), implementing the agent in a BDI-based agent programming language (section IV), and then adding explanation facilities to the implementation (section V). Although explanation facilities are added to the agent only in the end, the programming language and methods in the previous steps were chosen in such a way that this would be possible. In the paper we discuss all four steps, but we most explicitly explain how a task hierarchy can be translated to a BDI-based agent program. Section VI discusses related work, and in section VII, we draw conclusions and give suggestions

for future research.

II. Required behavior

To develop self-explaining agents for a virtual training system, knowledge about the required behavior and capacities of the agents is needed. Domain experts have knowledge about the tasks that belong to the jobs and roles of the agents, e.g. the tasks and responsibilities of a fire-fighter or an operator. Additionally, the training scenario(s) in which the agents will have to act give a lot of information about their required capacities. Training scenarios determine the scope of the situations in which the agents might arrive. For example, a firefighter might be responsible for the maintenance of tools, but tasks connected to this goal are not relevant in training scenarios about incidents, and thus do not have to be modeled.

One of the difficulties of writing a training scenario is to find a balance between freedom of the players (both agents and trainee) and continuation of a storyline, also called the *narrative paradox* [9]. On the one hand, trainees should be able to act as if it were a real situation and experience the consequences of their actions, e.g. believable reactions of the agents. On the other hand, because of inadequate acting of the trainee, the course of the scenario could change in such a way that situations in which specific learning goals can be trained do not occur. A possible solution of the narrative paradox in virtual training is to correct the trainee in a natural way if he deviates too much from the intended storyline, namely by directing the trainee with behavior of other players in the scenario. For instance, if a leading fire-fighter forgets to initiate smoke evacuation, initially nothing might happen, but eventually, a team member can ask for the smoke evacuation plan so that the trainee can practice to lead a smoke evacuation process. Thus, when determining the required capacities of an agent, tasks and actions involving the redirection of the trainee should be included.

A second aspect of importance for the scope of an agent's capacities concerns variation among different training scenarios. Most virtual training systems offer several training scenarios, to let trainees practice on different aspects of a mission. For instance, a fire-fighter might encounter incidents with or without victims, chemical substances or failing communication tools. Scenarios can be adjusted to the trainee's level of competence, e.g. scenarios are offered to the trainee with increasing difficulty. In conclusion, all possible scenario lines should be taken into account when determining the required capacities of an agent.

The required capacities of an agent in a scenario, including redirection actions and actions in different variations on the scenario, are input for the construction of a task hierarchy. A discussion on task hierarchies is given in the next section.

III. The agent's tasks

Writing a training scenario lays down the possible *observable* actions of an agent, but actions are the result of *unobservable* processes leading to select those particular actions. Many processes could underlie the generation of an action, but we believe the generation of behavior should be connected to behavior explanation. The deliberation steps that are taken to generate an action can also be best used to explain that action, and when these deliberation steps are understandable, the explanations should be as well. So while designing an agent with explanation capabilities, the unobservable internal processes should be meaningful.

In cognitive psychology, simple task analysis techniques restrict analyses to observable behavior, but cognitive task analysis also involves the knowledge, thought processes, and goal structures that underlie observable task performance. Hierarchical task analysis (HTA) is a well established technique [15], which smoothly connects observable behavior to internal cognitive processes by the systematical decomposition of goals or tasks. This feature makes it appropriate for developing self-explaining agents, who are supposed to explain the observable by the internal.

Many accounts of planning in artificial intelligence are based on hierarchical task representations, called hierarchical task networks (HTNs) [13]. In the strict sense, a HTN is the decomposition of an abstract task into more detailed subtasks; however, many accounts of HTN involve other features, e.g. information about which subtasks to select under given circumstances.

HTA and HTN planning both refer to a wide range of approaches, methods and techniques. In this paper, we leave it open *how* the task analysis should be made, but we do specify what should be the *result*. Namely, a task hierarchy represented in the task hierarchy representation language as introduced in the next section.

A. Task hierarchy representation language

A task hierarchy H in state S consists of a number of tasks which are related to each other by task-subtask relations. A task is defined as *T(Type,[(T1,C1),...,(Tn,Cn)])*, where *Type* denotes the type of task T, $[T1,..,Tn]$ are possible subtasks of task T, and $C1,..,Cn$ denote the conditions under which a subtask is adopted. Subtasks can in turn be decomposed into subtasks, etc. There are four types of tasks, namely *all*, *one*, *seq* and *prim*. Tasks of the type *all*, *one* or *seq* have subtasks, and a task's type denotes the relation to its subtasks. Tasks of the type *prim* are not decomposed, and can be achieved by executing a single action in the environment. In that case, the list with subtasks is empty: *T(prim,[])*.

Tasks can be adopted, which means that they are tried to achieve, and dropped, which means that they are either achieved or no longer tried to achieve. For all tasks but the top task it holds that a task can only be adopted when its main task is adopted, and the task is applicable, i.e. the conditions Ci connected to that task are consistent with state S. If there are no conditions connected to a task then $Ci = true$ and the task is always applicable. The three possible task-subtask relations *all*, *one* and *seq* each imply their own conditions to task adoption. A task-subtask relation of the type *all* means that all applicable subtasks are adopted. A task-subtask relation of the type *one* means that only one of the applicable subtasks

is adopted. The term *seq* refers to sequential and in such a task-subtask relation all subtasks are adopted, but only one by one in a specific order.

When a task is achieved it can be dropped. Task achievement depends either on state *S*, i.e. the conditions in the environment, or on achievement of a task's subtasks. An example of the first possibility is that the task to extinguish a fire is achieved when the fire is out. The second possibility occurs when the environment does not immediately give feedback on whether a task has been performed successfully. For instance, the task to report something can be achieved by sending an email, but one does not directly know if the email is read and understood. In such a cases task performance depends on task execution, that is, if a task is executed it is assumed to be achieved.

In our task hierarchy language, task achievement of primitive tasks always depends on task execution, and primitive tasks are thus always dropped after execution. Task achievement of non-primitive tasks either depends on conditions in state *S*, or on achievement of a task's subtasks. If the achievement of a task depends on the achievement of its subtasks, the relation with its subtasks defines the dependence relation. A task with a task-subtask relation of the type *all* is achieved when all applicable subtasks are achieved. A task with a task-subtask relation of the type *one* is achieved when exactly one of its subtasks is achieved. A task with a task-subtask relation of the type *seq* is achieved when all of its subtasks are achieved one by one, in the right order.

For each task holds that it either is achieved or not. There are several reasons for not allowing partially successful task executions. First, it is often hard to determine the measure of success of a task execution. Second, the task domains we aim at are of a procedural nature; if a task is not executed satisfactorily, alternative actions have to be taken, otherwise not. Third, the easiest way to represent partially succeeded tasks would be with a numerical approach, which makes it more difficult to provide explanations. The last two reasons together show that the domains we consider are appropriate for developing self-explaining agents.

B. An example: a firefighting agent

In this section we introduce an agent that could be one of the virtual characters in a scenario to train the head of a crisis management team. In such training, the trainee, who is playing the head, is confronted with a crisis he has to solve. A part of his tasks is to instruct and monitor the leaders of several teams, which are played by intelligent agents. The agent in this example is a firefighter, leading a firefighting team. The firefighting agent's tasks consist of receiving an attack plan from its head (the trainee) and pass corresponding instructions to its team, monitor the execution of the plan, and finally, report to the head that the incident has been solved.

Figure 1 shows the task hierarchy of the firefighting agent during the plan execution phase. Its two main occupations in this phase are checking the plan execution by its team, which involves dealing with a fire, victims and explosives, and reporting its observations to the head. Usually, the firefighter reports several times to the head, and it can report on one or more aspects of the incident at the same time. A representation of the task hierarchy in figure 1 in the task hierarchy representation language just introduced looks as follows.

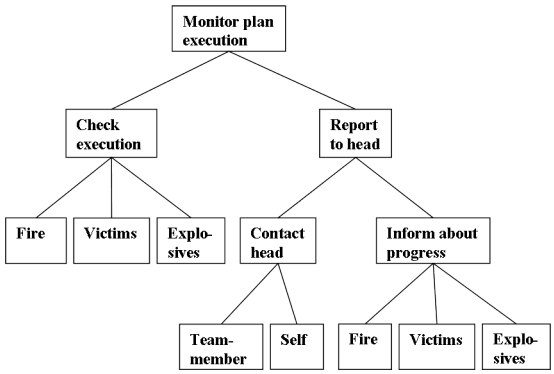

Fig. 1. Part of the firefighter's task hierarchy

```
Monitor(all,[(Check(X),true),(Report,true)])
Check(X)(all,[(Check(Fire),true),(Check(Victims),true),
      (Check(Explosives),true)])
Check(Fire)(prim,[])
Check(Victims)(prim,[])
Check(Explosives)(prim,[])
Report(seq,[(ContactHead(Y),torep(X)),(Inform(X),incontact)])
ContactHead(Y)(one,[(ContactHead(Member),available(member)),
      (ContactHead(Self),not available(member)])
ContactHead(Member)(prim,[])
ContactHead(Self)(prim,[])
Inform(X)(all,[(Inform(Fire),torep(Fire))),(Inform(Victims),
      torep(Victims)),(Inform(Explosives),torep(Explosives))])
Inform(Fire)(prim,[])
Inform(Victims)(prim,[])
Inform(Explosives)(prim,[])
```

In this representation the conditions *C* for task adoption and the task's relations *Type*, which are not shown in figure 1, are given. For example, *available(member)* means that there is a team member available, and *torep(X)* means that there is some information *X* that has not been reported to the head yet.

The *all* relation of the task Monitor denotes that the Check(X) and Report tasks are both adopted. To report, the firefighter has to contact the head and report about the different aspects of the incident. These two tasks have to be performed one by one and in this order, which is denoted by a *seq* relation. The fire-fighter agent can either contact the head itself, ContactHead(Self), or let a team member do it, ContactHead(Member). The *one* relation represents that executing one of these options should be sufficient to achieve contact with the head ContactHead(Y). Finally, the different aspects of the incident all have to be checked and reported, but the order in which they are checked and reported depends on how each of the aspects develops. This is denoted by an *all* relation.

IV. Implementation of the agent

The implementation of a self-explaining agent should fulfill four requirements. *First*, it should be possible to explicitly represent the agent's beliefs and goals. As stated in the introduction, the self-explaining agents should adopt the intentional stance to explain their behavior, i.e. explaining their actions in terms of goals. In section 3, we argued to connect the generation and explanation of behavior, e.g. if goal *G* makes an agent execute action *A*, *G* also explains why the agent executed *A*. Therefore, to let agents generate intentional explanations, they need to reason with beliefs and goals. The *second* requirement is that the operationalization of the agent's reasoning elements should be present in the implementation as well. Explanations should give insight into which elements relate to each other, how they relate to each other and how they interact. For instance, to achieve goal **G** it is necessary to achieve subgoal *Gi*. *Third*, a self-explaining agent should be able to introspect. An agent needs to have knowledge about its own states and processes in order to explain them. The *fourth* and last requirement is that a self-explaining agent needs to have memory. To explain its actions, an agent not only needs to have access to its states and processes at the time they occur, but also has to memorize them for possible future use.

Concerning the first requirement, a declarative agent programming language is needed. There are a number of agent programming languages based on the BDI (belief desire intention) model [12], which do allow for explicit representations of beliefs and goals. We have chosen to use 2APL [2] for our implementation as it connects declarative aspects like beliefs and goals to plans and actions, following from the interaction between beliefs and goals. Introspection is also possible in 2APL, the agent can perform checks on its belief and goal bases. A 2APL agent typically has no memory of its past states, processes and actions; these are only implicitly present in the interpreter. However, the agent's belief base can be considered as a memory, and a log about those aspects can be created in its belief base. For a more detailed overview of 2APL see [2].

A. Task hierarchies vs. BDI models

Our aim is to translate task hierarchy representations as discussed in section 3 to BDI-based agent programs. The most important similarity between task hierarchies and BDI models is that both reduce high-level entities into lower-level ones. A difference is that the first only contains tasks whereas the second makes a distinction between goals (desired world states) and plans (sequence of actions or sub-plans describing how a goal can be achieved). Table 1 shows the correspondence between the elements in a task hierarchy and a BDI-agent. An agent's main task must be implemented as a goal in order to generate plans and make the agent act, and primitive tasks can only be implemented as plans. However, the other tasks in the task hierarchy can be implemented as either plans or goals. In this section we discuss the advantages and disadvantages of representing tasks as goals or as plans.

Task hierarchy	**BDI agent**
State	Beliefs
Main task	Goal
Subtask	Goal or plan
Primitive task	Action (atomic plan)

TABLE I
Task Hierarchies vs. BDI agents

Beliefs and goals of an agent are related to each other in the sense that if an agent believes a certain fact, then the agent should not pursue that fact as a goal. In general, a goal remains in the agent's goal base until it is achieved, which is caused by sense actions leading to changes in the agent's belief base. In exceptional cases a goal can be explicitly dropped by the agent (as part of a plan). Plans, in contrast, are removed from an agent's plan base once they are executed. As a consequence, goals are more appropriate for the implementation of tasks which are achieved by an unknown number of actions (depending on the environment), e.g. monitoring plan execution.

Another difference between plans and goals concerns the way in which they are executed or achieved. The deliberation cycle of an agent states which step the agent should perform next, e.g. execute an action or apply a reasoning rule. In this cycle, the rules that are applicable to goals (PG-rules) are tried to be applied. But for plans, in contrast, plan by plan is considered which rules (PC-rules) apply to that plan. Thus, the order of goal execution depends on the order of the rules, whereas the order of plan execution depends on the order of the plans in the agent's plan base. As a programmer it is easier to exert control over the order of rules than over the order of plans in a plan base because an agent's rules remain the same, but its plans change during program execution.

For domains in which the number and order of tasks to be executed is fixed, it is easier to implement tasks as plans because the agent program ensures that plans are executed in the given order and dropped after execution automatically. In general, however, implementing tasks as goals allows for more flexibility because the number and order of actions to achieve a goal may vary. Therefore, we argue to implement all tasks in a hierarchy as goals, except for primitive tasks.

B. From task hierarchy to 2APL agent

In this section we discuss the translation of a task hierarchy representation as introduced in section III to 2APL code. In 2APL, an agent's beliefs are Prolog facts and rules, and the belief base of a 2APL agents is a Prolog program. Thus, from the beliefs *x* and *y :- x*, the belief *y* can be derived. To reason with goals, so called PG-rules are used, which are of the form Goal <− Belief | Plan. Informally this means that if the agent believes Belief, then to achieve the Goal it should execute Plan. To adopt a new goal (subtask), a Plan consists of the action *adopt(Subgoal)*, which means that the Subgoal is added to the agent's goal base. Dropping a goal is settled by beliefs in the agent's belief base. For each of the task types (all, one, seq, prim) we show the transition from task hierarchy representation to implementation. To ensure that the program 'walks through' the task hierarchy as desired,

we use the fact that the interpreter considers PG-rules from top to bottom. More specific rules are implemented above (and thus tried before) more general rules, so that the most specific rule as possible is applied.

a) All tasks: For tasks of the type *all*, all *applicable* subtasks are adopted. In the implementation, a PG-rule is added for each subtask, thus an *all*-task with n subtasks is implemented by n PG-rules. The task Monitor in our example has an *all* relation with its subtasks: Monitor(all, [(Check(X), true), (Report, true)]). The implementation looks as follows.

```
Monitor <- true | adopt(Check(X))
Monitor <- true | adopt(Report)
```

The first part of a rule is a check on the agent's goal base. Both PG-rules are only applied if Monitor is one of the agents goals. The second part of the PG-rule is a check on the agent's belief base. In this case, the guards of both rules are always true, i.e. the applicability of the subtasks is independent of the agent's beliefs. The body of the two rules states that the goals Check(X) and Report have to be adopted, respectively. If a subtask would only have to be adopted under certain circumstances, these conditions can be specified in the guard of the rule.

In 2APL, if a goal (a desired world state) is believed to be true, that goal is dropped. For some goals it holds that they are achieved if a certain situation in the environment is true. For example, the goal to extinguish a fire can be dropped when the agent believes that there is no fire.

```
ExtinguishFire :- not fire.
```

Other goals are achieved when its subtasks have been achieved. For those goals, to ensure that they are dropped when necessary, beliefs according to the following example should be in the agent's belief base.

```
Monitor :- Check(X), Report.
```

The code states that when both Check(X) and Report are finished successfully, the Monitor task can be dropped. If the goal Report would only be applicable under conditions C, the following rule would have to be added.

```
Report :- not C.
```

The rule ensures that if a subtask is not applicable (in the example when not C), that subtask does not have to be actively achieved in order to achieve its main task. Note that in some situations a task is achieved without executing any action, e.g. when there is no incident.

b) One tasks: For tasks of type *one* holds that only one of their subtasks is adopted. A *one*-task with n subtasks is implemented by n PG-rules. The task G(one,[(G1,C1),(G2,C2)]), for example, is implemented as follows.

```
G <- C1 | adopt(G1)
G <- C2 | adopt(G2)
```

C1 and C2 denote exclusive situations to ensure that only one sub-goal is adopted.

The goal G can be dropped either if a certain situation in the environment is true, or if one of the sub-goals is achieved. The last is implemented in the agent's belief base as follows.

```
G :- G1.
G :- G2.
```

Two separate beliefs are needed to express the dropping condition of goal G because G might be achieved by G1 or by G2.

c) Seq tasks: For tasks of the type *seq*, all of their subtasks are adopted, but one by one and in a specific order. For example, the task Report(seq, [(ContactHead(Y), torep(X)), (Inform(X), incontact)]) is implemented as follows.

```
Report <- torep(X)   | adopt(ContactHead(Y))
Report <- incontact | adopt(Inform(X))
```

The head of the rule contains the task for which subtasks need to be achieved. The guard of the rule contains conditions under which a rule can be adopted. As with tasks of the type *one*, the conditions specify unique circumstances here, so that only one subtask is executed at a time. Because the subtasks must be achieved in a specific order, the guards of the rules are beliefs related to goals, where the goal is the previous task in the sequence. For instance, the firefighter agent only starts to inform the head when it believes it is in contact. The belief torep(X) can be derived from the belief base if there is a priority that has been checked, but not yet been reported to the head, which is implemented as follows.

```
torep(X) :- Check(X), not Inform(X).
```

In general, a task with relation *seq* can be dropped if its last subtask is achieved.

```
Report :- Inform(X).
```

There is one exception. Namely, if the subtasks of a task of the type *seq* are primitive tasks, only one PG-rule is needed. For instance, a task with three subtasks who have to be executed one by one in a fixed order is implemented as follows.

```
Head <- Guard |
     { PrimTask1; PrimTask2; PrimTask3 }
```

As the order of the primitive subtasks is fixed and they can be executed immediately, it is not necessary to use different PG-rules. The different subtasks are added to the agent's plans base, and automatically executed in the right order.

d) Prim tasks: Primitive tasks are not divided into further subtasks and therefore implemented as plans. The following code shows an example of a task with the relation *one* to its primitive subtasks, namely ContactHead(Y)(one, [(ContactHead(Member), available(member)), (ContactHead(Self), not available(member)]).

```
ContactHead(Y) <- available(member) |
     ContactHead(Member)
ContactHead(Y) <- not available(member) |
     ContactHead(Self)
```

In this example the task ContactHead(Y) has two primitive subtasks. Instead of adopting a new goal, they can immediately be executed as actions in the environment. The implementation of primitive tasks with a parent task of the type *all* is similar to the example above.

As primitive tasks are implemented as plans, they are automatically removed from the agent's plan base once executed. If the goal for which they were executed is only dropped when certain conditions in the environment become true, they might be executed again. The other possibility is that their parent goal is dropped when (one of) its subtasks are (is) executed. In that case, the primitive task should involve an action in the environment, and add a belief to the agent's belief base which indicates that the action has been executed. In section V it will be explained how actual actions and belief update actions can be represented connected to each other, such that they are seen as one atomic action and their execution cannot be interrupted.

V. Explaining actions

As stated before, self-explaining agents should be able to introspect and memorize; they need to have knowledge about their own past states and processes in order to explain them. In 2APL, the information required for explanations is present in the program and the interpreter, but not available to the agent for explanation at a later moment in time. One possibility is to investigate how the information present in the interpreter could be made accessible for explanation purposes, but in this paper we chose to focus on a solution that does not require adaptations to the programming language.

In order to reproduce past actions and motivations, an agent needs to store them at the time it has access to them, which is during task execution. The agent can store its present beliefs, goals and actions in a so-called explanation log in its belief base. Such a log can be created by connecting belief update actions to actual actions of the agent. For instance, when the agent adopts goal G at time t, it also logs that it adopted goal G at t.

```
Monitor <- true |
     [ adopt(Check(X));
       UpdateLog(Check(X),t) ]
```

The [] brackets in the code ensure that the execution of the two plans *adopt()* and *UpdateLog()* can not be interrupted by any other process; they are considered as one atomic plan. Actions that update an agent's explanation log can be connected any 'normal' action of the agent. Such update actions can of course be connected to all actions of the agent, but some updates may not be needed in an explanation. Therefore, the decision what to log and what not should depend on the information that is desired in an explanation.

In order to explain its behavior, an agent needs a log of past actions, but it also needs to have knowledge about its own task hierarchy. This knowledge is represented by beliefs of the type **Task(T,[T1,..,Tn])**, where T1 to Tn are T's subtasks. The agent has such a belief for each task it could possibly adopt. For example, our firefighting agent has the following beliefs about its task hierarchy in its belief base.

```
Task(Inform(X),[Inform(Fire),
    Inform(Victims),Inform(Explosives)]).
Task(Report,[ContactHead(Y),Inform(X)]).
Task(Monitor,[Check(X),Report]).
```

With a combination of the beliefs containing the agent's complete task structure and beliefs that were logged during task execution, explanations of any action can be created. An extensive explanation is for example *I executed Inform(Victims) to achieve the goal Inform(X), which was executed to achieve the goal Report, which was executed to achieve the goal Monitor*. However, the complete trace of tasks responsible for one action might provide too much information; especially in bigger agent models it is crucial to make a selection of explaining elements. Such a selection consists of tasks either with a higher or a lower position in the hierarchy, yielding more abstract or specific explanations, respectively. An abstract explanation in this case would be: *I executed Inform(Victims) because I had the goal Monitor*. A specific explanation would be: *I executed Inform(Victims) because I had the goal Inform(X)*. More advanced explanation facilities could be interactive. For instance, the self-explaining agent starts with providing an abstract explanation, but if the trainee asks for extra information, more specific goals are provided.

VI. Related Work

Already at an early stage in expert systems research it was recognized that advices or diagnoses from decision support systems should be accompanied by explanations [17], [4]. An often made distinction is that between rule trace and justification explanations [17]. Rule trace or *how* explanations show which rules or data a system uses to reach a conclusion. Justification or *why* explanations, in addition, provide the domain knowledge underlying these rules and decisions. Research shows that users of expert system often prefer *why* to *how* explanations. An important difference between expert systems and self-explaining agents is that the self-explaining agents are proactive, i.e. they have goals. Therefore, the goals because of which an agent executed an action also form the explanation of that action.

Explanations in virtual training systems are often provided by intelligent tutoring systems (ITSs), for an overview see [10]. However, ITSs mostly provide hints and eventually recipes of what is to be done. There are a few approaches of self-explaining agents in virtual training systems that provide explanations from the agents' perspectives. The Debrief system [8] explains actions by what *must have been* an agent's underlying beliefs. However, the explanations do not provide

the agent's actual beliefs or goals. The XAI system [16] provides explanations about agents' physical states, e.g. position and health, but not about their motivations. An improved version of the XAI system [3] aims to provide explanations of agents' motivations by making use of information made available by the simulation. However, simulations often do not provide information about agents' motivations, and if so, the explanations are not based on the actual motivations of the agents.

Our approach of planning has similarities with HTN-planning approaches. Currently, one of the most extensive accounts of general (HTN) planning is the GPGP approach [7]). GPGP (generalized partial global planning) is a framework for the coordination of small teams of agents. Our approach differs on the following aspects with the GPGP approach. First, whereas the GPGP approach is designed for coordination of a group of agents, our approach is designed for planning of a single agent. Second, GPGP explicitly defines non-local task structures: relations between two goals at any place in the goal tree. In our model these relations are not specified, but implicitly present in the adoption conditions of the goals. For instance, the condition to adopt a goal is the achievement of another goal. Finally, the GPGP approach involves partially successful tasks, whereas we only allow tasks to be successful or not. The reasons for this choice are explained in the last paragraph of section III-A.

Sardina et al also used the similarities between BDI systems and HTN planners, for an approach on planning [14]. They present formal semantics for a BDI agent programming language which incorporates HTN-style planning as a built-in feature. In contrast to their approach, we do not incorporate HTN-planning in a BDI-based agent program; instead, we give a mapping of the former to the latter.

Self-explaining agents need to have knowledge about their past mental states and actions. Such knowledge is called an episodic or autobiographic memory. Most research on agents with an episodic memory focuses on how the memory can improve an agent's performance. In our approach, agents do not use their episodic memory during task execution, but only afterwards, to explain their behavior. Nevertheless, our approach has similarities with the work of Brom et al [1] as they also use a hierarchical structure to model their agents, both in the agent program and the episodic memory.

VII. Conclusions and future work

In this paper, we have introduced a methodology for developing self-explaining agents in virtual training systems. The methodology involves: i) a determination of the agent's required behavior, ii) the construction of a task hierarchy, iii) an implementation of the self-explaining agent in a BDI-based agent programming language, and iv) the addition of explanation capabilities.

Single phenomena and processes can be explained in many different ways, but providing complete explanations is neither possible, nor desired [5]. By using a BDI-based approach, the scope of possible explanations is restricted; actions are only explained in terms of beliefs and goals. Still, one action might have many underlying goals, and some selection on the information provided in an explanation might increase its effectiveness. For example, abstract explanations are given by just providing goals higher in the task hierarchy, and more specific explanations only consist of elements lower in the hierarchy. We are currently developing and implementing actual self-explaining agents for a specific virtual training system. When finished, we will be able to test the usefulness of the provided explanations, and investigate e.g. whether there is a general desired abstraction level of explanations.

Acknowledgments

This research has been supported by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO) and the Netherlands ICT Research and Innovation Authority (ICT Regie).

References

[1] C. Brom, K. Peskova, and J. Lukavsky. What does your actor remember? towards characters with a full episodic memory. In M. Cavazza and S. Donikian, editors, *Proc. of ICVS 2007*, pages 89–101. Springer-Verlag Berlin Heidelberg, 2007.

[2] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-agent Systems*, 16(3):214–248, 2008.

[3] D. Gomboc, S. Solomon, M. G. Core, H. C. Lane, and M. van Lent. Design recommendations to support automated explanation and tutoring. In *Proc. of the 14th Conf. on Behavior Representation in Modeling and Simulation*, Universal City, CA., 2005.

[4] S. Gregor and I. Benbasat. Explanation from intelligent systems: theoretical foundations and implications for practice. *MIS Quarterly*, 23(4):497–530, 1999.

[5] F. Keil. Explanation and understanding. *Annual Reviews Psychology*, 57:227–254, 2006.

[6] B. Keysar, S. Lin, and D. Barr. Limits on theory of mind use in adults. *Cognition*, 89:25–41, 2003.

[7] V. Lesser, K. Decker, N. Carver, A. Garvey, D. Neiman, M. Nagendra Prasad, and T. Wagner. Evolution of the GPGP/TAEMS domain-independent coordination framework. *Autonomous agents and muli-agent systems*, 9:87–143, 2004.

[8] W. Lewis Johnson. Agents that learn to explain themselves. In *Proc. of the 12th Nat. Conf. on Artificial Intelligence*, pages 1257–1263, 1994.

[9] S. Louchart and R. Aylett. Managing a non-linear scenario - a narrative evolution. In *Virtual Storytelling*, pages 148–157. Springer Berlin, 2005.

[10] T. Murray. Authoring intelligent tutoring systems: An analysis of the state of the art. *International Journal of Artificial Intelligence in Education*, 10:98–129, 1999.

[11] S. Nickerson. How we know -and sometimes misjudge- what others know: Imputing one's own knowledge to others. *Psychological Bulletin*, 125(6):737–759, 1999.

[12] A. Rao and M. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proc. of the 2nd Internat. Conf. on Principles of Knowledge Representation and Reasoning*, pages 473–484, San Mateo, CA, USA, 1991. Morgan Kaufmann publishers Inc.

[13] S. Russell and P. Norvig. *Artificial Intelligence A Modern Approach*. Pearson Education, Inc., New Jersey, USA, second edition, 2003.

[14] S. Sardina, L. De Silva, and L. Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proc. of AAMAS 2006*. ACM Press, 2006.

[15] J. Schraagen, S. Chipman, and V. Shalin, editors. *Cognitive Task Analysis*. Lawrence Erlbaum Associates, Mahway, New Jersey, 2000.

[16] M. Van Lent, W. Fisher, and M. Mancuso. An explainable artificial intelligence system for small-unit tactical behavior. In *Proc. of IAAA 2004*, Menlo Park, CA, 2004. AAAI Press.

[17] R. Ye and P. Johnson. The impact of explanation facilities on user acceptance of expert systems advice. *Mis Quarterly*, 19(2):157–172, 1995.