In: Proc. 26th German Conference on Artificial Intelligence (KI 2003). September 15 - 18, University of Hamburg, Germany, LNCS/LNAI,

Monitoring Agents using Declarative Planning ⋆

Jürgen Dix[1], Thomas Eiter[2], Michael Fink[2], Axel Polleres[2], and Yingqian Zhang[1]

[1] Department of Computer Science, University of Manchester, M13 9PL, UK
{dix,zhangy}@cs.man.ac.uk
[2] Institut für Informationssysteme, TU Wien, A-1040 Wien, Austria
{eiter,fink,polleres}@kr.tuwien.ac.at

Abstract. We present an *agent monitoring* approach, which aims at refuting from (possibly incomplete) information at hand that a multi-agent system (*MAS*) is implemented properly. In this approach, agent collaboration is abstractly described in an action theory. Action sequences reaching the collaboration goal are determined by a planner, whose compliance with the actual *MAS* behavior allows to detect possible collaboration failures. The approach can be fruitfully applied to aid offline testing of a *MAS* implementation, as well as online monitoring.
Keywords: knowledge representation, multi agent systems, planning

1 Introduction

Multi-Agent systems have been recognized as a promising paradigm for distributed problem solving, and numerous multi-agent platforms and frameworks have been proposed, which allow to program agents in languages ranging from imperative over object-oriented to logic-based ones [16]. A major problem which agent developers face with many platforms is verifying that a suite of implemented agents collaborate well to reach a certain goal (e.g., in supply chain management). Tools for automatic verification[3] are rare. Thus, common practice is geared towards extensive agent testing, employing tracing and simulation tools (if available).

In this paper, we present a *monitoring* approach which aids in automatically detecting that agents do not collaborate properly. In the spirit of Popper's *principle of falsification*, it aims at refuting from (possibly incomplete) information at hand that an agent system works properly, rather than proving its correctness. In our approach, agent collaboration is described at an abstract level, and the single steps in runs of the system are examined to see whether the agents behave "reasonable," i.e., "compatible" to a sequence of steps for reaching a goal.

Even if the internal structure of some agents is unknown, we may get hold of the messages exchanged among them. A given message protocol allows us to draw conclusions about the correctness of the agent collaboration. Our monitoring approach hinges on this fact and involves the following steps:

⋆ This work was supported by FWF (Austrian Science Funds), projects P14781 and Z29-N04, and partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-37004 WASP project.

[3] By well-known results, this is impossible in general but often also in simple cases if details of some agents (e.g., in a heterogenous environment) are missing.

(1) The intended collaborative behavior of the agents is modelled as a planning problem. More precisely, knowledge about the agent actions (specifically, messaging) and their effects is formalized in an *action theory*, T, which can be reasoned about to automatically construct *plans* as sequences of actions to reach a given goal.
(2) From T and the collaborative goal G, a set of intended plans, *I-Plans*, for reaching G is generated via a planner.
(3) The observed agent behavior, i.e., the message actions from a message log, is then compared to the plans in *I-Plans*.
(4) In case an incompatibility is detected, an error is flagged to the developer resp. user, pinpointing to the last action causing the failure so that further steps might be taken.

Steps 2-4 can be done by a special *monitoring agent*, which is added to the agent system providing support both in testing, and in the operational phase of the system. Among the benefits of this approach are the following:

- It allows to deal with collaboration behavior regardless of the implementation language(s) used for single agents.
- Depending on the planner used in step 2, different kinds of plans (optimal, conformant, ...), might be considered, reflecting different agent attitudes and collaboration objectives.
- Changes to the agent messaging by the system designer may be transparently incorporated to the action theory T, without further need to adjust the monitoring process.
- Furthermore, T adds to a formal system specification, which may be reasoned about and used in other contexts.
- As a by-product, the method may also be used for automatic *protocol generation*, i.e., determine the messages needed and their order, in a (simple) collaboration.

In the following, we detail the approach and illustrate it on an example derived from an implemented agent system. The next section describes the basic agent framework that we build upon and presents a (here simplified version) of a multi-agent system in the postal services domain. After that, in Section 3 we describe how to model the intended behavior of a multi-agent system as an abstract planning problem, and instantiate this for our example system using the action language $\mathcal{K}$ [5, 4]. Our approach to agent monitoring is then discussed in Section 4, where we also investigate some fundamental properties. After a brief discussion of the implementation in Section 5 and a review of related work in Section 6, we conclude in Section 7 with an outlook on further research.

2 Message Flow in a Multi-Agent System

In a multi-agent system (*MAS*), a set of autonomous agents are collaborating to reach a certain goal. Our aim is to monitor (some aspects of) the behavior of the agents in order to detect inconsistencies and help debugging the whole system.

As opposed to verification, monitoring a *MAS* does not require a complete specification of the behavior of the particular agents. Rather, we adopt a more general (and in practice much more realistic) view: We do not have access to the (entire) internal

state of a single autonomous agent, but we are able *to observe the communication between agents* of the system. By means of its communication capabilities, an agent can potentially control another agent. Our aim is to draw conclusions about the state of a multi-agent system by monitoring the message protocol.

2.1 Basic Framework

We consider multi-agent systems consisting of a finite set $\mathsf{A} = \{\mathtt{a}_1, \ldots, \mathtt{a}_n\}$ of collaborating agents $\mathtt{a}_i$. Although agents may perform a number of different (internal) actions, we assume that only one action is externally observable, namely an action called $\mathtt{send\_msg}(m)$, which allows an agent to send a message, m, to another agent in the system. Every `send_msg` action is given a timestamp and recorded in a message-log file containing the history of messages sent. The following definitions do not assume a sophisticated messaging framework and apply to almost any *MAS*. Thus, our framework is not bound to a particular *MAS*.

Definition 1 (Message, $\mathcal{M}_{log}$ file). *A* message *is a quadruple* $m = \langle \mathsf{s}, \mathsf{r}, \mathsf{c}, d \rangle$*, where* $\mathsf{s}, \mathsf{r} \in \mathsf{A}$ *are the identifiers of the* sending *and the* receiving *agents, respectively;* $\mathsf{c} \in \mathsf{C}$ *is from a finite set* C *of* message commands*;* d *is a list of constants representing the* message data*. A* message-log file *is an ordered sequence* $\mathcal{M}_{log} = t_1{:}m_1, t_2{:}m_2, \ldots, t_k{:}m_k$ *of messages* m_i *with timestamps* t_i*, where* $t_i \leq t_{i+1}$*,* $i < k$*.*

The set C constitutes a set of message *performatives* specifying the intended meaning of a message. In other words, it is the type of a message according to speech act theory: the illocutionary force of an utterance. These commands may range from ask/tell primitives to application specific commands fixed during system specification.

Often, an agent $\mathtt{a}_i$ will not send every kind of message, but use a message repertoire $\mathsf{C}_i \subseteq \mathsf{C}$. Moreover, only particular agents might be message recipients (allowing for simplified formats). Given that the repertoires C_i are pairwise disjoint and each message type c has a unique recipient, we use $\langle \mathsf{c}, d \rangle$ in place of $m = \langle \mathsf{s}, \mathsf{r}, \mathsf{c}, d \rangle$.

Finally, we assume a fixed bound on the time within the next action should happen in the *MAS*, i.e., a timeout for each action (which may depend on previous actions), which allows to see from $\mathcal{M}_{log}$ whether the *MAS* is stuck or still idle.

2.2 *Gofish* Post Office

We consider an example *MAS* called *Gofish Post Office* for postal services. Its goal is to improve postal product areas by mail tracking, customer notifications, and advanced quality control. The following scenario is our running example:

Example scenario: Pat drops a package, p_1, for a friend, Sue, at the post office. In the evening, Sue gets a phone call that a package has been sent. The next day, Sue decides to pick up the package herself at the post office on her way to work. Unfortunately, the clerk has to tell her that the package is already on a truck on its way to her home.

The overall design of the *Gofish MAS* is depicted in Figure 1. An *event dispatcher agent* (disp) communicates system relevant (external) events to an *event management agent* (em) which maintains an event database. Information about packages is stored

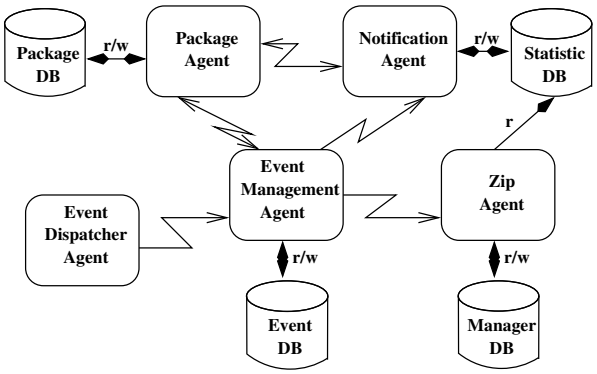

Fig. 1. The *Gofish* post office system.

in a package database manipulated by a *package agent* (pa). The *notification agent* (notify) notifies customers about package status and expected delivery time, for which it maintains a statistics database. Finally, a *zip agent* (zip) informs responsible managers, stored in a manager database, about zip codes not being well served.

Example 1 (Simple Gofish). For space reasons and to keep things simple and illustrative, we restrict the *Gofish MAS* to the package agent, pa, the event management agent, em, and the event dispatcher agent, disp; thus, $\mathsf{A} = \{\mathtt{pa}, \mathtt{em}, \mathtt{disp}\}$.

The event dispatcher informs the event manager agent about the drop off of a package (identified by a unique identifier), its arrival at the distribution center, its loading on a truck, its successful delivery, or when a recipient shows up at the distribution center to pick up a package by herself: $\mathsf{C}_{\mathtt{disp}} = \{\mathsf{dropOff}, \mathsf{distCenter}, \mathsf{truck}, \mathsf{delivery}, \mathsf{pickup}\}$. The event manager agent instructs the package agent to add a package to the package database after drop off, as well as to update the delivery time after delivery or customer pickup: $\mathsf{C}_{\mathtt{em}} = \{\mathsf{addPackage}, \mathsf{setDelivTime}\}$. The package agent here only receives messages, thus $\mathsf{C}_{pa} = \{\}$.
***Running scenario*:** The message-log $\mathcal{M}_{log}$ contains the messages $m_1 = \langle \mathsf{dropOff}, p_1 \rangle$, $m_2 = \langle \mathsf{addPackage}, p_1 \rangle$, $m_3 = \langle \mathsf{distCenter}, p_1 \rangle$, $m_4 = \langle \mathsf{truck}, p_1 \rangle$, and $m_5 = \langle \mathsf{pickup}, p_1 \rangle$. The entries are $0{:}\langle \mathtt{disp}, \mathtt{em}, \mathsf{dropOff}, p_1 \rangle$, $5{:}\langle \mathtt{em}, \mathtt{pa}, \mathsf{addPackage}, p_1 \rangle$, $13{:}\langle \mathtt{disp}, \mathtt{em}, \mathsf{distCenter}, p_1 \rangle$, $19{:}\langle \mathtt{disp}, \mathtt{em}, \mathsf{truck}, p_1 \rangle$, and $20{:}\langle \mathtt{disp}, \mathtt{em}, \mathsf{pickup}, p_1 \rangle$.

3 Modelling Agent Behavior via Declarative Planning

We now discuss how to formalize the intended collaborative agent behavior as an action theory for planning, which encodes the legal message flow. In it, actions correspond to messages and fluents represent assumptions about the current state of the world.

Under suitable encodings, we could use planning formalisms like STRIPS [8], PDDL [9] or HTN [7] based planners to model simple agent environments. In fact, HTN planning has recently been incorporated in a *MAS* [3] and formulated as action theories in logic programming [2]. Another powerful language suitable for modeling

control knowledge and plans for agents is GOLOG [15]. However, due to its high expressive power (loop, conditionals) automated plan generation is limited in this formalism. In Subsection 3.1 we give a generic formulation of our approach, independent of a particular planning mechanism. Then, in Subsection 3.2 we instantiate this high-level description using the action language $\mathcal{K}$ [5, 4]. While our approach does not rely on $\mathcal{K}$, we have chosen it because of its declarative nature and its capabilities to deal with incomplete knowledge and nondeterminism.

3.1 Modelling Intended Behavior of a *MAS*

Our approach to formalize the intended collaborative behavior of a *MAS* consisting of agents $\mathsf{A} = \{\mathtt{a}_1, \ldots, \mathtt{a}_n\}$ as a planning problem $\mathcal{P}$ comprises three steps:

Step 1: *Actions (Act).* Declare a corresponding *action* for each message $m = \langle \mathtt{s}, \mathtt{r}, \mathtt{c}, d \rangle$ in our domain, i.e., we have $\mathtt{c}(\mathtt{s}, \mathtt{r}, d) \in Act$ (see Def. 1). Again, if the message repertoires C_i are pairwise disjoint and each message type $\mathtt{c}$ has a unique recipient, we use in short $\mathtt{c}(d)$. These actions might have effects on the states of the agents involved and will change the properties that hold on them.
Step 2: *Fluents (Fl).* Define properties, *fluents*, of the "world" that are used to describe action effects. We distinguish between the sets of *internal* fluents, $Fl_{\mathtt{a}}$, of a particular agent $\mathtt{a}$[4] and *external* fluents, Fl_{ext}, which cover properties not related to specific agents. These fluents are often closely related to the message performatives C_i of the agents.
Step 3: *Theory (T)* and *Goal (G).* Using the fluents and actions from above, state various axioms about the collaborative behavior of the agents as a *planning theory* T. The axioms describe how the various actions change the state and under which assumptions they are executable. Finally, state the ultimate *Goal* G (in the running scenario: to deliver the package) suitable for the chosen planning formalism.

We end up with a *planning problem* $\mathcal{P} = \langle Act, Fl, T, G \rangle$, where $Fl = \bigcup_{\mathtt{a} \in \mathsf{A}} Fl_{\mathtt{a}} \cup Fl_{ext}$, whose *solutions* are a set of $\mathcal{P}$*-Plans*. Note that the precise formulation of these notions depends on the underlying planning formalism. For example, in HTN planning one has to specify *operators* and *methods* and their effects (this is closely related to *Act* and *Fl* above), as well as a domain description and a task list (which corresponds to T and G above): we refer to [2] for a full discussion. The above is a generic formulation suitable for many planning frameworks.

3.2 Using Action Language $\mathcal{K}$

In this section, we instantiate the planning problem $\mathcal{P}$ described above to a problem $\mathcal{P}^{\mathcal{K}}$ formulated in the action language $\mathcal{K}$. Due to space constraints, we only give the key features and refer to [5, 4] for further details.

Declarations of the type $p(\overline{X})\ requires\ bk_1(\overline{Y_1}), \ldots, bk_m(\overline{Y_m})$ define *actions* and *fluents* p in $\mathcal{K}$ ($\overline{X} = X_1, \ldots, X_n$ are parameters that must be *typed* by some predicates

[4] Internal fluents especially can describe private values which might be inaccessible by an external observer.

$bk_1, \ldots, bk_m$ defined in the so called *background knowledge* BK, which specifies a finite set of static facts in a function-free first-order language). In addition, $\mathcal{K}$ allows to state axioms of the following form:[5]

(1) `caused` f `if` α `after` β.
(2) `total` f `if` α `after` β.
(3) `inertial` f.
(4) `executable` a `if` β.
(5) `nonexecutable` a `if` β.

(1) means that fluent f is caused whenever α holds after β. (2) simulates nondeterministic effects: its meaning is that fluent f is either true or false if α holds after β. (3) models inertia of a fluent f: it is a macro for `caused` f `if not` $\neg.f$ `after` f, where `not` is default negation and $\neg$ is strong negation. Furthermore, with (4) and (5) we can express under which circumstances an action is *executable* or *nonexecutable*.

A planning problem $\mathcal{P}^{\mathcal{K}}$ may then be formalized as a tuple $\langle Act, Fl, T, G\rangle$, where *Act* defines the actions, *Fl* the fluents, T comprises BK and all axioms (of the sort introduced above), and G is the goal, i.e. a set of ground fluent literals.

The semantics of $\mathcal{K}$ is defined through *legal transitions* $t = \langle s, A, s'\rangle$ from states s to states s' by simultaneous execution of actions A, where a *state* s is any consistent set of ground fluent literals.[6] A *trajectory* Tr is any initial state s_0 or sequence $t_1, \ldots, t_n$ of legal transitions $t_i = \langle s_{i-1}, A_i, s_i\rangle$, $i \in \{1, \ldots, n\}$, starting in an initial state s_0. An *(optimistic) plan* for goal G is $P = \langle\rangle$, resp. the projection $P = \langle A_1, \ldots, A_n\rangle$ of a trajectory Tr, such that G holds in s_0 resp. s_n.

Example 2 (Simple Gofish cont'd). In the *Gofish* example, the following $\mathcal{K}$ actions (corresponding to the possible messages) and fluents are defined (in $\text{DLV}^{\mathcal{K}}$ notation [4]):

```
actions : dropOff(P) requires pkg(P).
          addPkg(P) requires pkg(P).
          distCenter(P) requires pkg(P).
          truck(P) requires pkg(P).
          delivery(P) requires pkg(P).
          pickup(P) requires pkg(P).
          setDelivTime(P) requires pkg(P).
```

Act

```
fluents : pkgAt(P, Loc) requires pkg(P), loc(Loc).
          delivered(P) requires pkg(P).
          recipAtHome(P) requires pkg(P).
          added(P) requires pkg(P).
          delivTimeSet(P) requires pkg(P).
```

Fl

The first three external fluents describe the current location of a package, whether it has successfully been delivered, and whether its recipient is at home, respectively. The last two fluents are internal fluents about the state of agent `pa`; whether the package has already been added to the package database resp. whether the delivery time has been set properly.

[5] In all the statements below, f is a fluent literal, a is an action, α is a set of (possibly default negated) fluent literals, and β is a set of (possibly default negated) actions and fluent literals.

[6] Note that in $\mathcal{K}$ states are not "total", i.e., a fluent f can be neither true nor false in a state.

A possible package (e.g., a generic p_1) and its locations are background knowledge represented by the set of facts BK=$\{pkg(p_1), loc(drop), loc(dist), loc(truck)\}$. Now we specify further axioms for T (in $\text{DLV}^{\mathcal{K}}$ notation) as follows:

```
initially : recipAtHome(p1).
always :    noConcurrency.
 inertial pkgAt(P, L).       inertial delivered(P).
 inertial recipAtHome(P).    inertial added(P).

 executable dropOff(P) if not added(P).
 caused pkgAt(P, drop) after dropOff(P).
 nonexecutable dropOff(P) if pkgAt(P, drop).

 executable addPkg(P) if pkgAt(P, drop), not added(P).
 caused added(P) after addPkg(P).

 executable distCenter(P) if added(P), pkgAt(P, drop).
 caused pkgAt(P, dist) after distCenter(P).
 caused -pkgAt(P, drop) after distCenter(P).

 executable truck(P) if pkgAt(P, dist), not delivered(P).
 caused pkgAt(P, truck) after truck(P).
 caused -pkgAt(P, dist) after truck(P).

 executable delivery(P) if pkgAt(P, truck),  not delivered(P).
 caused delivered(P) after delivery(P), recipAtHome(P).

 executable setDelivTime(P, DTime) if delivered(P).
 caused delivTimeSet(P) after setDelivTime(P).

 executable pickup(P) if pkgAt(P, dist),  not delivered(P).
 executable pickup(P) if pkgAt(P, truck),  not delivered(P).
 caused delivered(P) after pkgAt(P, dist),  pickup(P).
 total recipAtHome(P) after pickup(P).
```

Most of the theory is self-explanatory. The recipient is at home initially. The keyword `noConcurrency` specifies that concurrent actions are disallowed. An important aspect is modelled by the final `total` statement. It expresses uncertainty whether after a pickup attempt at the distribution center, the recipient will be back home, in particular in time before the truck arrives to deliver the package, if it was already on the way. Finally, the goal is G = `delivTimeSet`(p_1).

The following (optimistic) plans reach G:

$$P_1 = \langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{truck}(p_1); \texttt{pickup}(p_1); \texttt{delivery}(p_1); \texttt{setDelivTime}(p_1)\rangle$$
$$P_2 = \langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{truck}(p_1); \texttt{delivery}(p_1); \texttt{setDelivTime}(p_1)\rangle$$
$$P_3 = \langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{pickup}(p_1); \texttt{setDelivTime}(p_1)\rangle$$

In P_1, the recipient shows up at the distribution center after the package is loaded on the truck and the truck is on its way. In P_2, the package is successfully delivered before the recipient comes to pick it up herself, whereas in P_3, she picks up the package before it has been loaded on the truck.

Running scenario: According to the message history in $\mathcal{M}_{log}$, we see that plan P_2 is infeasible, as well as P_3 since the package can not be handed over to Sue at the distribution center. Thus, only P_1 remains for successful task completion.

4 Agent Monitoring

The overall aim of adding a monitoring agent (monitor) is to *aid debugging a given MAS*. We can distinguish between two principal types of errors: *(1) design errors*, and *(2) implementation (or coding) errors*. While the first type means that the model of the system is wrong (i.e., the *MAS* behaves correctly to the model of the designer of the *MAS*, but this model is faulty and does not yield the desired result in the application), the second type points to more mundane mistakes in the actual code of the agents: the code does not implement the formal model of the system (i.e., the actions are not implemented correctly).

Note that often it is very difficult, if not impossible at all, to distinguish between design and implementation errors. But even before the system is deployed, the planning problem $\mathcal{P}$ can be given to a planner and thus the overall existence of a solution can be checked. If there is no solution, this is clearly a design error and the monitoring agent can pinpoint where exactly the planning fails (assuming the underlying planner has this ability). If there are solutions, the agent designer can check them and thus critically examine the intended model.

However, for most applications the bugs in the system become apparent only at runtime. Our proposed monitoring agent has the following structure.

Definition 2 (Structure of the monitoring agent). *The agent* monitor *loops through the following steps:*

1. *Read and parse the message log $\mathcal{M}_{log}$. If $\mathcal{M}_{log} = \emptyset$, the set of plans for $\mathcal{P}$ may be cached for later reuse.*
2. *Check whether an action timeout has occurred.*
3. *If this is not the case, compute the current* intended plans *(according to the planning problem description and additional info from the designer) compatible with the actions as executed by the* MAS.
4. *If no compatible plans survive, or the system is no more idle, then inform the agent designer about this situation.*
5. *Sleep for some pre-specified time.*

We now elaborate more deeply into these tasks.

Checking *MAS* behavior: monitor continually keeps track of the *messages sent between the agents*. They are stored in the message-log, $\mathcal{M}_{log}$, which is accessible by monitor. Thus for monitor, the behavior of the *MAS* is completely determined by $\mathcal{M}_{log}$. We think this is a realistic abstraction from internal agent states. Rather than describing all the details of each agent (which might be unknown, e.g. if legacy agents are involved), the kinds of messages sent by an agent can be chosen so as to give a declarative high-level view of it. In the simplified *Gofish* example, these messages for agents em, disp, pa are given by $\mathsf{C}_{\mathtt{em}}$, $\mathsf{C}_{\mathtt{disp}}$, and $\mathsf{C}_{\mathtt{pa}}$ (see Section 2).

Intended behavior and compatibility: The desired collaborative *MAS* behavior is formalized as a planning problem $\mathcal{P}$ (e.g., in language $\mathcal{K}$, cf. Section 3). Thus, even before the *MAS* is in operation, problem $\mathcal{P}$ can be fed into a planner which computes potential plans to reach a goal. Agent monitor is exactly doing that.

In general, not all $\mathcal{P}$-Plans may be admissible, as constraints may apply (derived from the intended collaborative behavior). [7] E.g., some actions ought to be taken in fixed order, or actions may be penalized with costs whose sum must stay within a limit. We thus distinguish a set *I-Plans*$(\mathcal{P}) \subseteq \mathcal{P}$*-Plans* as *intended plans* (of the *MAS* designer).

It is perfectly possible that the original problem has successful plans, yet after some actions executed by the *MAS*, these plans are no longer valid. This is the interesting case for the agent designer since it clearly shows that something has gone wrong: monitor can pinpoint to the precise place indicating which messages have when caused the plan to collapse. Because these messages are related to actions executed by the agents, information about them will help to debug the *MAS*. In general, it is difficult to decide whether the faulty behavior is due to a coding or design error. However, the info given by monitor will aid the agent designer to detect the real cause.

Messages from monitor: Agent monitor continually checks and compares the actions taken so far for compatibility with all current plans. Once a situation has arisen in which no successful plan exists (detected by the planner employed), monitor writes a message into a separate file containing (1) the first action that caused the *MAS* to go into a state where the goal is not reached, (2) the sequence of actions taken up to this action, and (3) all the possible plans *before* the action in 1) was executed (these are all plans compatible with the *MAS* behavior up to it).

In the above description, we made heavily use of the notion of a *compatible* plan. Before giving a formal definition, we consider our running scenario. In *Gofish*, all three plans P_1, P_2, P_3 generated from the initial problem coincide on the first three steps: dropOff(p_1), addPkg(p_1), and distCenter(p_1).

Running scenario (coding error): Suppose on a preliminary run of our scenario, $\mathcal{M}_{log}$ shows m_1=dropOff(p_1). This is compatible with each plan P_i, $i \in \{1, 2, 3\}$. Next, m_2=distCenter(p_1). This is incompatible with each plan; monitor detects this and gives a warning. Inspection of the actual code may show that the command for adding the package to the database is wrong. While this doesn't result in a livelock (the *MAS* is still idle), the database was not updated. Informed by monitor, this is detected at this stage already.

After correction of this coding error, the *MAS* may be started again and another error shows up:

Running scenario (design error): Instead of waiting at home (as in the "standard" plan P_2), Sue shows up at the distribution center and made a pickup attempt. This "external" event may have been unforeseen by the designer (problematic events could also arise from *MAS* actions). We can expect this in many agent scenarios: we have no complete knowledge about the world, unexpected events may happen, and action effects may not fully determine the next state.

[7] This might depend on the capabilities of the underlying planning formalism to model constraints such as cost bounds or optimality wrt. resource consumption etc.

Only plan P_1 remains to reach the goal. However, there is *no guarantee of success*, if Sue is not back home in time for delivery. This situation can be easily captured in the framework of [5, 4]. There, we have the notion of a *secure* plan. An (optimistic) plan is *secure* (or *conformant* [11]), if regardless of the initial state and the outcomes of the actions, the steps of the plan will always be executable one after the other and reach the goal (i.e., in all trajectories). As can be easily seen, P_1 is not secure. Thus, a design error is detected, if delivering the package must be guaranteed under any circumstances. Based on a generic planning problem $\mathcal{P}$, we now define compatible plans as follows.

Definition 3 ($\mathcal{M}_{log}$ compatible plans). *Let the planning problem $\mathcal{P}$ model the intended behavior of a* MAS*, which is given by a set I-Plans$(\mathcal{P}) \subseteq \mathcal{P}$-Plans. Then, for any message log $\mathcal{M}_{log} = t_1{:}m_1, \ldots, t_k{:}m_k$, we denote by C-Plans$(\mathcal{P}, \mathcal{M}_{log}, n)$, $n \geq 0$, the set of plans from I-Plans$(\mathcal{P})$ which comply on the first n steps with the actions $m_1, \ldots, m_n$.*

Respecting that the $\mathcal{K}$ planner, $\texttt{DLV}^{\mathcal{K}}$, is capable of computing optimistic and secure plans, we denote for any $\mathcal{K}$ planning problem $\mathcal{P}^{\mathcal{K}}$ by X-Plans$^o(\mathcal{P}^{\mathcal{K}}, \mathcal{M}_{log}, n)$ (resp. X-Plans^s $(\mathcal{P}^{\mathcal{K}}, \mathcal{M}_{log}, n)$ the set of all optimistic (resp. secure) plans for $\mathcal{P}^{\mathcal{K}}$ with the above property, $X \in \{\mathsf{I}, \mathsf{C}\}$.

Definition 4 (Culprit$(\mathcal{M}_{log}, \mathcal{P})$). *Let $t_n{:}m_n$ be the first entry of $\mathcal{M}_{log}$ such that either (i) C-Plans$(\mathcal{P}, \mathcal{M}_{log}, n) = \emptyset$ or (ii) a timeout is detected. Then, Culprit$(\mathcal{M}_{log}, \mathcal{P})$ is the pair $\langle t_n{:}m_n, idle\rangle$ if (i) applies and $\langle t_n{:}m_n, timeout\rangle$ otherwise.*

Initially, $\mathcal{M}_{log}$ is empty and thus C-Plans$(\mathcal{P})$ = I-Plans$(\mathcal{P})$. As more and more actions are executed by the *MAS*, they are recorded in $\mathcal{M}_{log}$ and the set C-Plans$(\mathcal{P})$ shrinks. monitor can thus compare at any point in time whether C-Plans$(\mathcal{P}, \mathcal{M}_{log}, n)$ is empty or not. Whenever this happens, Culprit$(\mathcal{M}_{log}, \mathcal{P})$ is computed and pinpoints to the problematic action.

Running scenario: Under guaranteed delivery (i.e., secure planning), monitor writes Culprit$(\mathcal{M}_{log}, \mathcal{P})$=$\langle 20{:}m_5, idle\rangle$ (the $pickup(p_1)$ message) on a file, and thus clearly points to a situation missed in the *MAS* design. Note that there are also situations where everything is fine; if pickup would not occur, agent monitor would not detect a problem at this stage.

4.1 Properties

We can show that the agent monitoring approach has desirable properties. The first result concerns its soundness.

Theorem 1 (Soundness). *Let the planning problem $\mathcal{P}$ model the intended collaborative* MAS *behavior, given by I-Plans$(\mathcal{P}) \subseteq \mathcal{P}$-Plans. Let $\mathcal{M}_{log}$ be a message log. Then, the* MAS *is implemented incorrectly if Culprit$(\mathcal{M}_{log}, \mathcal{P})$ exists.*

Semantically, the intended collaborative *MAS* behavior (described in any formalism) may manifest in a set of trajectories as described for $\mathcal{K}$ planning problems, where trajectories correspond to possible runs of the *MAS* (sequences of states and executed actions). On the other hand, optimistic plans for a $\mathcal{K}$ planning problem $\mathcal{P}^{\mathcal{K}}$ are projected

trajectories. We say that a set OP of such plans *covers* the intended collaborative *MAS* behavior, if each run of the *MAS* corresponds to some trajectory whose projection is in OP. For example, this holds if OP is the set of all optimistic plans for $\mathcal{P}^{\mathcal{K}}$ and the intended collaborative *MAS* behavior is given by a secure plan, or, more liberally, by a conditional plan. We have:

Theorem 2 (Soundness of $\mathcal{P}^{\mathcal{K}}$ Cover). *Let $\mathcal{P}^{\mathcal{K}}$ be a $\mathcal{K}$ planning problem, such that* I-Plans$^{o}(\mathcal{P}^{\mathcal{K}})$ *covers the intended collaborative* MAS *behavior. Let $\mathcal{M}_{log}$ be a message log. Then,* MAS *is implemented incorrectly if* Culprit$(\mathcal{M}_{log}, \mathcal{P}^{\mathcal{K}})$ *exists.*

As for completeness, we need the assertion that plans can not grow arbitrarily long, i.e., have an upper bound on their length.

Theorem 3 (Completeness). *Let the planning problem $\mathcal{P}$ model the intended collaborative* MAS *behavior, given by* I-Plans$(\mathcal{P}) \subseteq \mathcal{P}$-Plans *where plans are bounded. If the* MAS *is implemented incorrectly, then there is some message log $\mathcal{M}_{log}$ such that either (i)* C-Plans$(\mathcal{P}, \mathcal{M}_{log}, 0) = \emptyset$*, or (ii)* Culprit$(\mathcal{M}_{log}, \mathcal{P})$ *exists.*

In (i), we can conclude a design error, while in (ii) a design or coding error may be present. There is no similar completeness result for $\mathcal{P}^{\mathcal{K}}$ covers; note that in our running scenario, a design error is detected for secure plans as *MAS* collaborative behavior formalism. However, the culprit vanishes if the cover contains plan P_1, which is compatible with $\mathcal{M}_{log}$.

As for complexity, we mention that in expressive planning formalisms like $\mathcal{K}$, deciding whether C-Plans$(\mathcal{P}, \mathcal{M}_{log}, n) \neq \emptyset$ or Culprit$(\mathcal{M}_{log}, \mathcal{P})$ exists from $\mathcal{P}$, $\mathcal{M}_{log}$ and n is NP-hard in general, which is inherited from expressive planning language. We remind that, like for satisfiability (SAT), this is a theoretical worst-case measure, though, and still solutions for many instances can be found quickly. Moreover, there are instance classes which are polynomial time solvable and for which DLV$^{\mathcal{K}}$ is guaranteed to compute plans in polynomial time.

5 Implementation

To demonstrate the proposed approach, a running example has been implemented. The *Gofish MAS* and Agent monitor is developed within IMPACT (*Interactive Maryland Platform for Agents Collaborating Together*). Note that in principle our approach is completely independent of any specific agent system. We refer to [17] for the details of IMPACT.

Each agent consists of a set of *data types*, *API functions*, *actions*, and an *agent program* that includes some rules prescribing its behaviors. Since we use DLV$^{\mathcal{K}}$ [4] as the planner, a new connection module has been created within Agent monitor so that monitor can access the DLV$^{\mathcal{K}}$ planner. In this way, before the *Gofish MAS* operates, we feed $\mathcal{P}^{\mathcal{K}}_{Gofish}$ into monitor, which then exploits DLV$^{\mathcal{K}}$ to compute all potential plans including both secure and optimistic plans.

Running scenario: The *Gofish* post office guarantees the package delivery within 24 hours of dropOff (time 0). Consider the case that Sue wanted to pick up her package

(p_1=0x00fe6206c.1) at the distribution center. Unfortunately, it has been loaded on the truck. Sue did not come back home in time, therefore the package wasn't delivered in time. Thus after the "pickup" action at time 20, the *MAS* was keeping idle till a timeout (24 in this example) was detected by monitor. In the end, monitor generated a log file as follows (see also the project webpage [8]):

Problematic action:
20:pickup($0x00fe6206c.1$), timeout

Actions executed:
0:dropOff($0x00fe6206c.1$); 5:addPkg($0x00fe6206c.1$);
13:distCenter($0x00fe6206c.1$); 19:truck($0x00fe6206c.1$)

Possible plans before problematic action:
⟨dropOff(p_1); addPkg(p_1); distCenter(p_1); truck(p_1);
pickup(p_1); delivery(p_1); setDelivTime(p_1)⟩
⟨dropOff(p_1); addPkg(p_1); distCenter(p_1); truck(p_1);
delivery(p_1); setDelivTime(p_1)⟩

6 Related Work

Monitoring problem has been raised in Robotics literature. [10] presented a situation calculus-based account of execution monitoring for robot programs. A situation calculus specification is given for the behavior of a Golog program. The interpretation of Golog programs is combined with an execution monitor, which detects and recovers from discrepancies. Similar to our method, their approach is formal and works for monitoring arbitrary programs. While we focus on monitoring the collaboration of multiple agents, they address the problem of a single agent acting in an uncertain environment.

Another interesting monitoring approach is based on multi-agent *plan-recognition*, by Tambe [18], Intille and Bobick [12], Devaney and Ram [1], Kaminka et al. [13, 14]. In this approach, an agent's intentions (goals and plans), beliefs or future actions are inferred through observations of another agent's ongoing behavior.

Devaney and Ram [1] describe the plan recognition problem in a complex multi-agent domain involving hundreds of agents which act over large space and time scales. They use pattern matching to recognize team tactics in military operations. The team-plan library stores several strategic patterns which the system needs to recognize during the military operation. In order to make computation efficient, they utilize representations of agent-pair relationships for team behaviors recognition.

Intille and Bobick [12] constructed a probabilistic framework that can represent and recognize complex actions based on visual evidence. Complex multi-agent action is inferred using a multi-agent belief network. The network integrates the likelihood values generated by several visual goal networks at each time and returns a likelihood that a given action has been observed. The network explicitly represents the logical and temporal relationships between agents, and its structure is similar to a naive Bayesian classifier network structure, reflecting the temporal structure of a particular complex

[8] http://www.cs.man.ac.uk/~zhangy/project/monitor/

action. Their approach relies on all *coordination constraints* among the agents. Once an agent fails, it may not be able to recognize the plans.

Another line of work has been pursued in ISI. Gal Kaminka et al. [13, 14] developed the *OVERSEER* monitoring system, which builds upon work on multi-agent plan-recognition by [12] and [18]. They address the problem of many geographically distributed team members collaborating in a dynamic environment. The system employs plan recognition to infer the current state of agents based on the observed messages exchanged between them. The basic component is a *probabilistic plan-recognition algorithm* which underlies the monitoring of a single agent and runs separately for each agent. This algorithm is built under a Markovian assumption and allows linear-time inference. To monitor multiple agents, they utilize social knowledge, i.e. relationships and interactions among agents, to better predict the behavior of team members and detect coordination failures. *OVERSEER* supports reasoning about *uncertainty* and *time*, and allows to answer queries related to the likelihood of current and future team plans.

Comparison: While our objective is (1) to debug *offline* an implemented MAS, and (2) to monitor *online* the collaboration of multiple agents, the approaches described above mainly aim to inferring (sub-)team plans and future actions of agents. They do not address the MAS debugging issue. Furthermore, we point out that our method might be used in the MAS design phase to support *protocol generation*, i.e., determine at design time the messages needed and their order, for a (simple) agent collaboration. More precisely, possible plans $P = \langle m_1, \ldots, m_k \rangle$ for a goal encode sequences of messages m_1,..., m_k that are exchanged in this order in a successful cooperation achieving the goal. The agent developer may select one of the possible plans, e.g. according to optimality criteria such as least cost, P^*, and program the individual agents to obey the corresponding protocol. In subsequent monitoring and testing, P^* is then the (single) intended plan.

Plan recognition, which is adopted for multi-agent monitoring by the above approaches, is suitable for various situations: if communication is *not* possible, agents exchanging messages are not reliable, or communications must be secure.

The above methods significantly differ from our approach in the following points:

(1) If a multi-agent system has already been deployed, or it consists of legacy code, the plan-recognition approach can do monitoring without modifications on the deployed system. Our method entirely relies on an agent message log file.
(2) The algorithms developed in [14] and [1] have low computational complexity. Especially the former is a linear-time plan recognition algorithm.
(3) Our model is not yet capable of reasoning about uncertainty, time and space.
(4) In some tasks, agents do not frequently communicate with others during task execution. In addition, communication is not always reliable and messages may be incorrect or get lost.

We believe the first three points can be taken into account in our framework. (1) Adding an agent actions log file explicitly for a given MAS should not be too difficult. (2) While the developed algorithms are of linear complexity, the whole framework needs to deal with uncertainty or probabilistic reasoning which can be very expensive. While our approach is NP-hard in the worst case, we did not encounter any difficulties

in the scenarios we have dealt with. (3) Although IMPACT does not yet have implemented capabilities for dealing with probabilistic, temporal and spatial reasoning, such extensions have been developed and are currently being implemented.

Among the advantages of our method are the following:

- Our method can be more easily extended to do *plan repair* than the methods above. Merely Kaminka et al. mentioned the idea of dealing with failure actions.
- The approach we have chosen includes protocol generation in a very intuitive sense relying on the underlying planner while the cited approaches model agent behavior at an abstract level which can not be used to derive intended message protocols directly.
- Since ascertaining the intentions and beliefs of the other agents will result in uncertainty with respect to that information, some powerful means of reasoning under uncertainty are required for the plan recognition method.

7 Conclusion

We have described a method to support testing of a multi-agent system, based on monitoring their message exchange using planning methods. This can be seen as a very useful debugging tool for detecting coding and design errors. We also presented some soundness and completeness results for our approach, and touched its complexity.

Our approach works for arbitrary agent systems and can be tailored to any planning formalism that is able to express the collaborative behavior of the *MAS*. We have briefly discussed (and implemented) how to couple a specific planner, $\texttt{DLV}^{\mathcal{K}}$, which is based on the language $\mathcal{K}$, to a particular *MAS* platform, viz. IMPACT. A webpage for further information and detailed documentation has been set up (see footnote 8).

There are many extensions to our approach. We mention just some:

(1) Cost based planning: Can the goal still be reached with a certain bound on the overall costs, given that actions which the agents take have costs attached? And, what is the optimal cost and how does a corresponding behavior look like? This would allow us to assess the quality of an actual agents behavior and to select cost-effective strategies. To keep the exposition simple, we have omitted that $\texttt{DLV}^{\mathcal{K}}$ is also capable of computing admissible plans (plans within a cost bound) and, moreover, optimal plans over optimistic and secure plans, respecting that each action has certain declared cost [6]. For instance, in the *Gofish* example we might prefer plans where the customer picks up the package herself, which is cheaper than sending a truck. Thus, in the realization of our approach, also economic behavior of agents in a *MAS* under cost aspects can be easily monitored, such as obedience to smallest number of message exchanges or least total communication cost.

(2) Dynamic planning: We assumed an *a priori* chosen collaboration plan for $\mathcal{M}_{log}$ compatibility. This implies $\mathsf{C\text{-}Plans}(\mathcal{P}, \mathcal{M}_{log}, n') \subseteq \mathsf{C\text{-}Plans}(\mathcal{P}, \mathcal{M}_{log}, n)$, for all $n' \geq n \geq 0$. However, this no longer holds if the plan may be dynamically revised. Checking $\mathcal{M}_{log}$ compatibility then amounts to a new planning problem whose initial states are the states reached after the actions in $\mathcal{M}_{log}$.

(3) At the beginning of monitoring, all potentially interesting plans for the goal are generated, and they can be cached for later reuse. We have shown the advantages of this

method. However, if a very large number of intended plans exists up front, the method may become infeasible. In this case, we might just check, similar as above, whether from the state reached and the actions in $\mathcal{M}_{log}$, the goal can be reached.

Investing the above issues is part of ongoing and planned future research.

References

1. M. Devaney and A. Ram. Needles in a haystack: Plan recognition in large spatial domains involving multiple agents. In *Proc. AAAI-98*, pp. 942–947, 1998.
2. J. Dix, U. Kuter, and D. Nau. HTN planning in answer set programming. Tech. Rep. CS-TR-4332, CS Dept., Univ. Maryland, 2002. Submitted to *Theory and Practice of Logic Programming*.
3. J. Dix, H. Munoz-Avila, and D. N. and Lingling Zhang. Theoretical and empirical aspects of a planner in a multi-agent environment. In G. Ianni and S. Flesca, editors, *Proc. European Conference on Logic and AI (JELIA '02)*, LNCS 2424, pp. 173–185. Springer, 2002.
4. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: the DLV$^{\mathcal{K}}$ system. *Artificial Intelligence*, 144(1-2):157–211, 2003.
5. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Transactions on Computational Logic*, 2003. To appear.
6. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Answer Set Planning under Action Costs. *Journal of Artificial Intelligence Research*, 2003. To appear.
7. K. Erol, J. A. Hendler, and D. S. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In K. J. Hammond, editor, *Proc. 2nd Int'l Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pp. 249–254. AAAI Press, 1994.
8. R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.
9. M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL — The Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control, October 1998. Available at `http://www.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz`.
10. G.D. Giacomo, R. Reiter and M. Soutchanski. Execution Monitoring of High-Level Robot Programs. In *Principles of Knowledge Representation and Reasoning*, pp. 453–465, 1998.
11. R. Goldman and M. Boddy. Expressive planning and explicit knowledge. In *Proc. AIPS-96*, pp. 110–117. AAAI Press, 1996.
12. S. S. Intille and A. F. Bobick. A framework for recognizing multi-agent action from visual evidence. In *Proc. AAAI-99*, pp. 518–525, 1999.
13. G. Kaminka, D.V.Pynadath, and M. Tambe. Monitoring deployed agent teams. In *Proc. Fifth International Conference on Autonomous Agents (Agents-2001)*, pp. 308–315. ACM, 2001.
14. G. A. Kaminka and M. Tambe. Robust agent teams via socially-attentive monitoring. *Journal of Artificial Intelligence Research*, 12:105–147, 2000.
15. H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31(1–3):59–83, 1997.
16. M. Luck, P. McBurney, C. Preist, and C. Guilfoyle. The Agentlink agent technology roadmap draft. Available at `http://www.agentlink.org/roadmap/index.html`, 2002.
17. V.S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems: Theory and Implementation*. MIT Press, 2000.
18. M. Tambe. Tracking dynamic team activity. In *Proc. AAAI-96*, pp. 80–87, 1996.